

# MR-CUDASW – GPU ACCELERATED SMITH-WATERMAN ALGORITHM FOR MEDIUM-LENGTH (META)GENOMIC DATA

A Thesis Submitted to the  
College of Graduate Studies and Research  
in Partial Fulfillment of the Requirements  
for the degree of Master of Science  
in the Department of Computer Science  
University of Saskatchewan  
Saskatoon

By

Amir Muhammadzadeh

©Amir Muhammadzadeh, July/2014. All rights reserved.

## PERMISSION TO USE

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science  
176 Thorvaldson Building  
110 Science Place  
University of Saskatchewan  
Saskatoon, Saskatchewan  
Canada  
S7N 5C9

# ABSTRACT

The idea of using a graphics processing unit (GPU) for more than simply graphic output purposes has been around for quite some time in scientific communities. However, it is only recently that its benefits for a range of bioinformatics and life sciences compute-intensive tasks has been recognized. This thesis investigates the possibility of improving the performance of the overlap determination stage of an Overlap Layout Consensus (OLC)-based assembler by using a GPU-based implementation of the Smith-Waterman algorithm.

In this thesis an existing GPU-accelerated sequence alignment algorithm is adapted and expanded to reduce its completion time. A number of improvements and changes are made to the original software. Workload distribution, query profile construction, and thread scheduling techniques implemented by the original program are replaced by custom methods specifically designed to handle medium-length reads.

Accordingly, this algorithm is the first highly parallel solution that has been specifically optimized to process medium-length nucleotide reads (DNA/RNA) from modern sequencing machines (i.e. Ion Torrent). Results show that the software reaches up to 82 GCUPS (Giga Cell Updates Per Second) on a single-GPU graphic card running on a commodity desktop hardware. As a result it is the fastest GPU-based implementation of the Smith-Waterman algorithm tailored for processing medium-length nucleotide reads. Despite being designed for performing the Smith-Waterman algorithm on medium-length nucleotide sequences, this program also presents great potential for improving heterogeneous computing with CUDA-enabled GPUs in general and is expected to make contributions to other research problems that require sensitive pairwise alignment to be applied to a large number of reads. Our results show that it is possible to improve the performance of bioinformatics algorithms by taking full advantage of the compute resources of the underlying commodity hardware and further, these results are especially encouraging since GPU performance grows faster than multi-core CPUs.

## ACKNOWLEDGEMENTS

Foremost, I would like to express my deepest thanks to my supervisor, Dr. Kusalik. His patience, encouragement, and immense knowledge were key motivations throughout my study. His forensic scrutiny of my technical writing has been invaluable. He has always found the time to propose consistently excellent improvements. I am truly thankful for his dedication to both my personal and academic development.

Special thanks to my committee members: Dr. Ian McQuillan, Dr. Mark Keil and Dr. Luis Rueda for their helpful comments and suggestions. Special thanks to my loving siblings Nazanin and Aria for their unwavering support and encouragement throughout my education.

Some of the content of this thesis could not have been included without the help of my colleagues. Specifically, Dr. Brett Trost was instrumental in gathering the results presented in Chapter 4. He also generously provided suggestions and comments that are incorporated in this study.

Finally, I would be remiss if I did not thank my loving partner, Michelle. Nearly everything I have done for the last couple of years is here. Pain and excitement are here, as well as feeling good and bad, the despair and the indescribable joy of accomplishment. And on top of these are all the gratitude and love I have for her.

To the living memory of my mother ...

# CONTENTS

<b>Permission to Use</b>	<b>i</b>
<b>Abstract</b>	<b>ii</b>
<b>Acknowledgements</b>	<b>iii</b>
<b>Contents</b>	<b>v</b>
<b>List of Tables</b>	<b>vii</b>
<b>List of Figures</b>	<b>viii</b>
<b>List of Abbreviations</b>	<b>x</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Aim of the thesis . . . . .	3
1.2 Structure of this document . . . . .	5
<b>2 Research goal</b>	<b>6</b>
2.1 Research goals . . . . .	6
2.2 Limitations . . . . .	7
<b>3 Background</b>	<b>8</b>
3.1 Sequence assembly . . . . .	8
3.1.1 Brief overview of existing assembly strategies . . . . .	9
3.1.2 Overlap-layout-consensus strategy . . . . .	9
3.1.3 MIRA: an automated genome and EST assembler . . . . .	10
3.2 Sequence alignment . . . . .	11
3.2.1 Principles of sequence alignment . . . . .	11
3.2.2 Scoring alignments and substitution matrices . . . . .	13
3.2.3 Dynamic programming algorithms . . . . .	14
3.2.4 Types of alignment . . . . .	14
3.2.5 Algorithmic approximations . . . . .	19
3.3 Parallel computing . . . . .	20
3.3.1 CPU vs. GPU . . . . .	24
3.4 GPU computing . . . . .	26
3.4.1 Why CUDA? . . . . .	26
3.4.2 CUDA architecture . . . . .	28
3.4.3 Kepler vs. Fermi architecture . . . . .	31
<b>4 Comparison of assembly software for metagenomic data</b>	<b>33</b>
4.1 Methods . . . . .	33
4.1.1 Artificial metagenomic communities . . . . .	33
4.1.2 Generation of artificial reads . . . . .	34
4.2 Results . . . . .	34
4.3 Conclusions . . . . .	42
<b>5 Data and methodology</b>	<b>43</b>
5.1 Structure of this chapter . . . . .	43

5.2	Selection of the fittest . . . . .	45
5.2.1	Related works . . . . .	45
5.2.2	GPU-accelerated sequence aligners . . . . .	47
5.2.3	Comparing GPU-accelerated sequence alignment tools . . . . .	48
5.2.4	Is CUDASW++ 3.0 fast enough? . . . . .	49
5.3	Improving the fittest . . . . .	50
5.3.1	CUDASW++ 3.0 . . . . .	50
5.4	MR-CUDASW . . . . .	56
5.4.1	Sequence length deviation & thread scheduling . . . . .	56
5.4.2	Query profile . . . . .	61
5.4.3	Ensuring the fidelity of the result . . . . .	63
<b>6</b>	<b>Results</b>	<b>64</b>
6.1	Benchmarking GPU-accelerated Smith-Waterman tools . . . . .	64
6.1.1	Metric . . . . .	65
6.1.2	Benchmarking . . . . .	65
6.2	Improving CUDASW++ 3.0 . . . . .	67
6.2.1	Sequence length deviation and thread scheduling . . . . .	68
6.2.2	Query profile . . . . .	69
6.3	Evaluating MR-CUDASW . . . . .	74
<b>7</b>	<b>Conclusion and discussion</b>	<b>77</b>
7.1	Conclusion and remarks . . . . .	77
7.2	Discussion . . . . .	80
7.3	Future work . . . . .	81
	<b>References</b>	<b>82</b>
<b>A</b>	<b>Comparison of assembly software</b>	<b>87</b>
A.1	Evaluating various assembly software using low complexity simulated dataset . . . . .	87
A.2	Evaluating various assembly software using medium complexity simulated dataset . . . . .	88
A.3	Evaluating various assembly software using high complexity simulated dataset . . . . .	89
A.4	Evaluating various assembly software using real WGS dataset . . . . .	90
<b>B</b>	<b>Core PTX SIMD assemblies</b>	<b>91</b>
<b>C</b>	<b>Performance details of GPU-accelerated alignment tools</b>	<b>92</b>

# LIST OF TABLES

3.1	GPU Computing Applications . . . . .	27
3.2	Major differences between Kepler and Fermi architectures . . . . .	32
4.1	List of evaluated sequence assemblers . . . . .	35
4.2	Characteristics of input data . . . . .	35
4.3	CPU details of various assembly software (simLC) . . . . .	37
4.4	CPU details of various assembly software (simMC) . . . . .	39
4.5	CPU details of various assembly software (simHC) . . . . .	40
4.6	CPU details of various assembly software, assembling real WGS data. . . . .	42
5.1	Characteristics of input data . . . . .	49
6.1	Performance details of CUDASW++ 3.0 and MIRA (simulated datasets) . . . . .	70
6.2	Performance details of CUDASW++ 3.0 and MIRA (real WGS datasets) . . . . .	71
A.1	Size statistics of various assembly programs (simLC) . . . . .	87
A.2	Percentage of contigs produced by various assembly software (simLC) . . . . .	87
A.3	Size statistics of various assembly programs (simMC) . . . . .	88
A.4	Percentage of contigs produced by various assembly software (simMC) . . . . .	88
A.5	Size statistics of various assembly programs (simHC) . . . . .	89
A.6	Percentage of contigs produced by various assembly software (simHC) . . . . .	89
A.7	Size statistics of various assembly programs with real WGS data . . . . .	90
C.1	Performance details of GPU-accelerated alignment software tools (SYN_1000.fna) . . . . .	92
C.2	Performance details of GPU-accelerated alignment software tools (SYN_10K.fna) . . . . .	92
C.3	Performance details of GPU-accelerated alignment software tools (SYN_100K.fna) . . . . .	92
C.4	Performance details of GPU-accelerated alignment software tools (ENV_1000.fna) . . . . .	93
C.5	Performance details of GPU-accelerated alignment software tools (ENV_10K.fna) . . . . .	93
C.6	Performance details of GPU-accelerated alignment software tools (ENV_100K.fna) . . . . .	93
C.7	Performance details of CUDASW++ 3.0, MR-CUDASW ,MIRA, and water (simulated data)	94
C.8	Performance details of CUDASW++ 3.0, MR-CUDASW ,MIRA, and water (real WGS) . . .	94



# LIST OF FIGURES

1.1	Phases of a MIRA assembly cycle . . . . .	4
3.1	The general principle of sequence alignment . . . . .	12
3.2	Gapping preserves the maximum similarity between two sequences . . . . .	12
3.3	There is never just one possible sequence alignment between any two sequences . . . . .	13
3.4	A global alignment may be viewed as a path through a directed path graph . . . . .	15
3.5	An optimal alignment for two sequences using local alignment technique . . . . .	17
3.6	Serial computation . . . . .	20
3.7	Parallel computation . . . . .	21
3.8	Single Instruction, Single Data (SISD) organization . . . . .	22
3.9	Single Instruction, Multiple Data (SIMD) organization . . . . .	22
3.10	Multiple Instruction, Single Data (MISD) organization . . . . .	23
3.11	Multiple Instruction, Multiple Data (MIMD) organization . . . . .	24
3.12	Differences between CPU and GPU architectures . . . . .	25
3.13	Automatic scalability of CUDA architecture . . . . .	29
3.14	Grid of thread blocks . . . . .	30
3.15	Memory hierarchy of CUDA . . . . .	32
4.1	Size statistics of various assembly programs (simLC) . . . . .	37
4.2	Percentage of contigs produced by various assembly software (simLC) . . . . .	38
4.3	Size statistics of various assembly programs (simMC) . . . . .	38
4.4	Percentage of contigs produced by various assembly software (simMC) . . . . .	39
4.5	Size statistics of various assembly programs (simHC) . . . . .	40
4.6	Percentage of contigs produced by various assembly software (simHC) . . . . .	41
4.7	Size statistics of various assembly programs with real data . . . . .	41
5.1	Methodology flow chart . . . . .	44
5.2	Program workflow of CUDASW++ 3.0 . . . . .	51
5.3	Data independences in the alignment matrix . . . . .	54
5.4	Vector arrangement techniques . . . . .	55
5.5	Program workflow of MR-CUDASW . . . . .	57
5.6	CUDA thread synchronization rules . . . . .	58
5.7	Arrangement of subject sequences in the database for the inter-task parallelization . . . . .	59
5.8	The workload distribution technique employed in the improved version of CUDASW++ 3.0 . . . . .	60
5.9	Approaches to vectorization of Smith-Waterman alignments . . . . .	62
5.10	Data dependencies between SIMD registers holding the H values with the Rognes and Wozniak implementations. . . . .	63
6.1	Performance details of GPU-accelerated alignment software tools (simulated datasets) (GT 640) . . . . .	66
6.2	Performance details of GPU-accelerated alignment software tools (real WGS datasets) (GT 640) . . . . .	67
6.3	Performance details of GPU-accelerated alignment software tools (simulated datasets) (GTX 680) . . . . .	68
6.4	Performance details of GPU-accelerated alignment software tools (real WGS datasets) (GTX 680) . . . . .	69
6.5	Effects of modifications on improving the performance of CUDASW++ 3.0 (simulated data) . . . . .	71
6.6	Effects of modifications on improving the performance of CUDASW++ 3.0 (real WGS data) . . . . .	72
6.7	Effects of modifications on improving the performance of CUDASW++ 3.0 (simulated data) . . . . .	72
6.8	Effects of modifications on improving the performance of CUDASW++ 3.0 (real WGS data) . . . . .	73
6.9	Effects of modifications on improving the performance of CUDASW++ 3.0 (simulated data) . . . . .	74
6.10	Effects of modifications on improving the performance of CUDASW++ 3.0 (real WGS data) . . . . .	75

6.11 Performances of GPU-accelerated alignment software tools and MIRA (simulated data) . . .	76
6.12 Performances of GPU-accelerated alignment software tools and MIRA (real WGS data) . . .	76

# LIST OF ABBREVIATIONS

API	Application Programming Interface
BCUPS	Billion Cell Updates per Second
BLOSUM	BLOck SUBstitution Matrix
bp	base pair
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
Cell/BE	Cell Broadband Engine Architecture
DNA	DeoxyriboNucleic Acid
EBI	European Bioinformatics Institute
EMBL	European Molecular Biology Laboratory
FPGA	Field-Programmable Gate Array
FSB	Front Side Bus
GCUPS	Giga Cell Updates per Second
GPL	General Public License
GPU	Graphics Processing Unit
GPGPU	General-Purpose Computing on Graphics Processing Units
HGT	Horizontal Gene Transfer
ISA	Instruction Set Architecture
MIMD	Single-instruction, Multiple-thread
MISD	Multiple-instruction, Single-thread
MPI	Message Passing Interface
NGS	Next-Generation Sequencing
OLC	Overlap-layout-consensus
OpenGL	Open Graphics Library
OS	Operating System
PAM	Percent Accepted Mutations
PBSM	Per-block Shared Memory
PHAT	Predicted Hydrophobic A Transmembrane matrix
PGM	Personal Genome Machine
PS3	PlayStation 3
PTX	Parallel Thread Execution
PTLM	Per Thread Local Memory
RAM	Random Access Memory
RNA	RiboNucleic Acid
SIMD	Single-Instruction, Multiple-Data
SIMT	Single-instruction, Multiple-thread
SISD	Single-instruction, Single-thread
SLIM	ScoreMatrix Leading to Intra-Membrance
SM	Streaming Multiprocessor
SP	Scalar Processor
SSE	Streaming SIMD Extensions
SW	Smith-Waterman
WGS	Whole Genome Sequencing

# CHAPTER 1

## INTRODUCTION

Since Frederick Sanger and colleagues sequenced the first genome in 1977 [1], genetic sequencing technologies have significantly improved. The advent of faster and cheaper sequencing technologies has accelerated biological and biomedical research dramatically and has led to such fields as metagenomics.

Metagenomics provides an unbiased and broad insight into the microbial world. A vast amount of microbial sequencing data is being generated through large-scale projects in ecology (e.g. environmental genome shotgun sequencing of the Sargasso Sea [2]), agriculture (e.g. promoting healthier humans through healthier livestock using metagenomics [3]), and human health (e.g. human microbiome project [4]). The analyses of community samples have provided a new way of examining the microbial world that not only has changed the landscape of environmental microbiology but also has paved the road to a better understanding of the entire living world [5]. For these reasons, this new field of study has been compared to the reinvention of the microscope [6].

The recent developments in post-Sanger sequencing technologies, commonly referred to as next-generation sequencing (NGS), have substantially shaped the way metagenomics studies are performed. Today, the sequencing of a metagenome (the combined genome of all organisms in an ecological community) with billions of base pairs in length is done routinely regardless of the organisms' ability to be cultured in the laboratory. Accordingly, the massive majority of organisms ( $\sim 99\%$  of microbes) that classic microbiology fails to culture can be analyzed in metagenomics studies in an inexpensive and high-throughput manner [7]. However, unlike Sanger sequencing that results in read lengths of  $\sim 800$  bp, the currently available NGS technologies generate much shorter reads:  $\sim 50$ - $75$  bp (Applied Biosciences/Life Technologies SOLiD),  $\sim 75$ - $150$  bp (Solexa/Illumina Sequencing by Synthesis),  $100$ - $200$  bp (IonTorrent/Life Technologies Semiconductor Chip Sequencing) and  $400$ - $600$  bp (454/Roche Pyrosequencing)[8]. The downstream analysis of such short read data is a major obstacle; one of the biggest challenges is whole genome assembly, a problem that is still far from being solved.

DNA sequencing data from NGS platforms typically present shorter read lengths compared with Sanger sequencing data. Higher coverage, and different error profiles are also two other differences that can be found in the data generated by these platforms. Since 2005, several assembly software packages have been developed specifically for assembly of next-generation sequencing data. However not all assemblers can be utilized for assembly of metagenomics projects. Unlike a genome project that ultimately aims to determine the complete genome sequence of a single organism, in studying metagenomic data thousands of genomes from

an entire microbial community are studied simultaneously. Genomes for most of these organisms do not exist in public bioinformatics databases, so it is not possible to assemble the reads by mapping them to a reference genome. As a result, reference-free *de novo* assembly must be applied. The most serious problem for *de novo* assembly of metagenomics are genomic diversity and variable abundance within populations. Additionally, inadequate and partial sampling of different species' genomes along with presence of repetitive fragments from innumerable complicated genomes are other factors that burden reconstructing the full metagenome [9].

The information content of a sequence has been found to be highly dependent on the sequence length. Accordingly, although a full assembly of a metagenome might not be attainable, aligning and merging reads in order to construct progressively longer contiguous sequences (contigs) is still beneficial. Longer fractions of genomes let us find open reading frames, operons, operational transcriptional units, their associated promoter elements, and transcription factor binding sites. Furthermore, longer elements such as pathogenicity islands, and other mobile genetic elements, are evident only when large fractions of the genome are assembled [9].

Whether genetic materials are recovered from environmental samples or from cultivated clonal cultures, the choice of assembly strategy highly depends on the sequencing technology that is being used, which in return essentially correlates to accessibility and budget constraints. As outlined by Dear et al. [10], a sequence assembly is essentially a set of contigs, each contig being the consensus of a multiple alignment of reads. Unfortunately, the underlying problem of string assembly as a variant of the shortest common superstring problem has been shown to be NP-hard [11]. Nevertheless, many tools have already been developed to address this question such as MIRA [12] and Newbler [13] assembly software tools. These software share a common paradigm, sometimes referred to as overlap-consensus-layout [14]. This approach is quite similar to the one generally used when solving a jigsaw puzzle [15]. The first step consists of aligning the fragments two-by-two in an exhaustive fashion and establishing which pairs of reads present a consistent overlap with one another. This is analogous to searching for pieces of the puzzle which fit each other and have matching colours. The main difficulty at this stage is to distinguish erroneous overlaps due to sequencing errors and those due to similarities within the genome, such as highly conserved repeats [16].

The assembler then detects clusters of reads which consistently align with each other, thus forming contiguous sequences (or contigs). This is equivalent to having parts of the image put together in a puzzle. In both genome and puzzle assembly, the process is interrupted at either ambiguous areas, where several continuations are possible, or at gaps, where no connecting piece has been found. Finally, the assembler attempts to order and orient the contigs with respect to one another. Returning to the puzzle simile, this corresponds to placing the corners and identifiable parts of the image relative to each other. Using for example paired-end information, the assembler can estimate the distance that separates contigs. Sets of contigs which can all be plausibly placed together in the same region are sometimes called scaffolds or supercontigs [15].

All the currently existing assemblers can broadly be classified, based on the data structures they are using, into three categories, all based on graphs. The Overlap/Layout/Consensus (OLC) methods rely on an overlap graph. The de Bruijn Graph (DBG) methods use some form of K-mer graph. The greedy graph

algorithms may use OLC or DBG [17].

Although traditional assemblers have been, and will be, used in metagenomic projects, the assembly of a metagenome is different from the assembly of a single genome. Unlike genomic assemblers, in which the fundamental problem is handling repetitive genomic fragments that often lead to misassembly, metagenomic assemblers are expected to surmount the additional challenges posed by inhomogeneous organism abundances within a sampled community, horizontal gene transfer (HGT) events between co-existence species, different complexities in terms of the overall number of organisms contained and the presence of multiple closely related organisms.

As the preliminary stage of this thesis, an evaluation of six popular *de novo* assembly software tools was conducted (the results of this evaluation is presented in Chapter 4). The accuracy, performance, and computational requirements of these assemblers were evaluated using three datasets of simulated sequence reads, as well as real reads obtained from the sequencing of environmental samples using Ion Torrent technology. Unfortunately no single assembler performed best on all our criteria, leaving the question of what is the best assembler for medium-length metagenomic reads unsolved. However, our comparison of assembly software tools showed that OLC-based assemblers are still the best choice for assembling medium-length metagenomic reads. According to these results MIRA, although slow, slightly outperformed the other participating assembly software tools. It scored very well in size statistics and correctness analysis while producing a very large aggregated contig size.

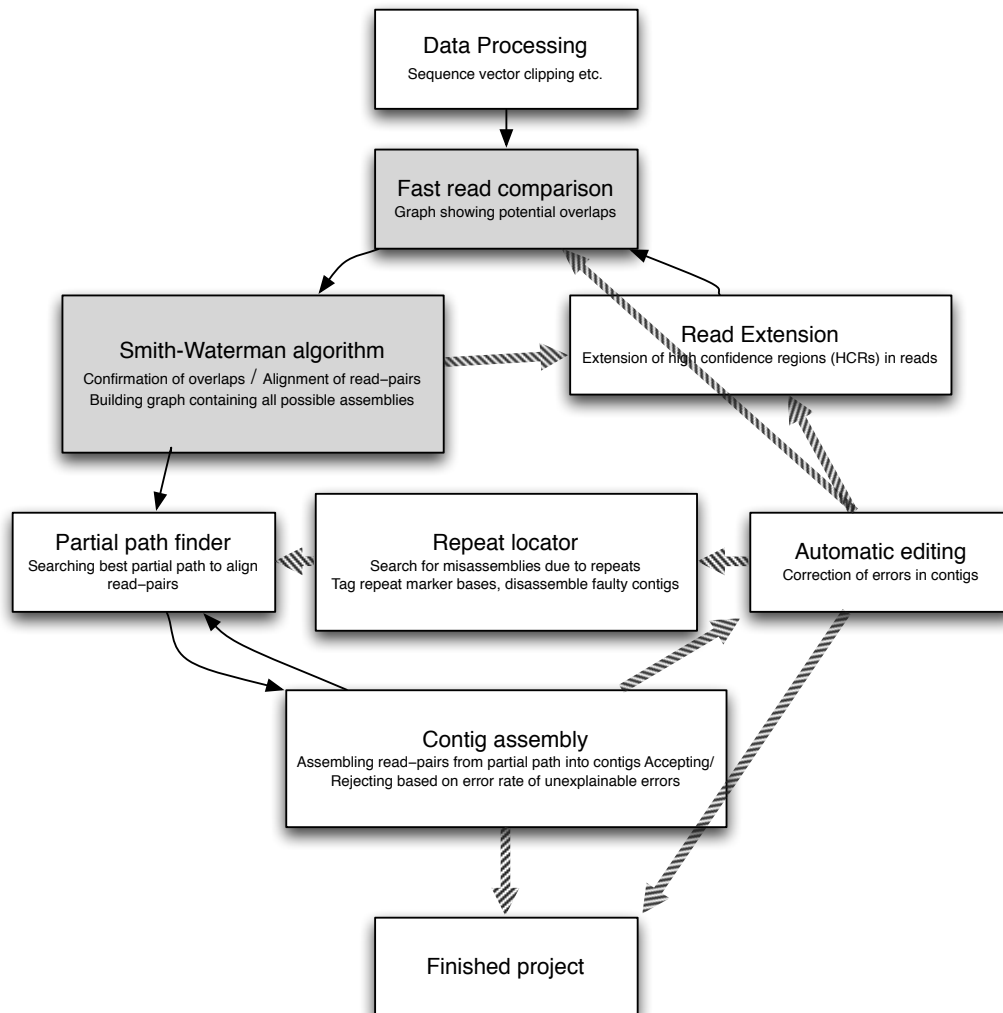
## 1.1 Aim of the thesis

Considering the computational time, maximum random access memory (RAM) occupancy, assembly accuracy and integrity, and the presence of programs' source and its maintainability and modularity, our study identified MIRA as the best potential assembly software that could meet our performance expectations while having grounds for improvement and modifications. To this end, several independent sequential modules implemented in MIRA (Figure 1.1) could be replaced with scalable replacements. Specifically, the most compute-intensive portion of the algorithm (i.e. the alignment step) can be modified to exploit the distributed computing capacity of the available hardware and take advantages of GPU and/or additional CPUs. Although dynamic programming techniques are commonly used for computing optimal pairwise sequence alignments, their corresponding complexities are quadratic with respect to the lengths of alignment targets [18], which makes them time consuming for applications involving large datasets. Therefore heuristic methods have been introduced in literatures to accelerate sequence alignment. The drawback is that the more computationally efficient the heuristics, the worse the quality of the result. Another approach to get high-quality results in a short time is to use high-performance computing.

In the course of this thesis work we will discuss how the emergence of accelerator technologies and many-core architectures, such as FPGAs, Cell/BEs and GPUs, has provided the opportunity to significantly

reduce the runtime for many bioinformatics programs including the Smith-Waterman algorithm on commonly available and inexpensive hardware. This study seeks a GPU-accelerated computing solution which uses a graphics processing unit (GPU) together with a CPU, as an alternative computing solution to massively distributed solutions, to accelerate the alignment step of the assembly process. It is expected that such an updated assembler can highlight the potential and effectiveness of multi/many-core computing as a viable option to achieve significant speedup with high efficiency in assembling large and complex metagenomic datasets.

**Figure 1.1:** Phases of a MIRA assembly cycle. Plain arrows show imperative pathways, dashed arrows denote optional pathways. This figure is taken from Chevreux [12].



## 1.2 Structure of this document

As stated above, this work seeks an alternative computing solution to massively distributed solutions, often employed by other assemblers (e.g. RayMèta [19]), that require sophisticated parallel tools which are not accessible to every ordinary genomic laboratory. This study offers a GPU-accelerated computing solution to accelerate the alignment step of the assembly process. It is expected that this approach can provide a large level of parallelism using a fraction of the budget required by massively distributed computing solutions.

The rest of this thesis is organized as follows. Chapter 2 summarizes the goals of this research and describes the limitations this thesis work is subjected to. Background to the concepts presented in this thesis is given in Chapter 3. Chapter 4 presents the results of the evaluation of *de novo* assemblers for metagenomic data that was done as part of this thesis. Chapter 5 describes the design and the implementation, as well as the data and methods used to evaluate it against the sequential implementation of the SW algorithm, the technique used by MIRA assembler. Chapter 6 presents and evaluates the results. Finally, Chapter 7 gives some concluding remarks, discuss some issues relating to the results, as well as few topics that could be further investigated in order to improve on or extend this work.



# CHAPTER 2

## RESEARCH GOAL

The major objective of this thesis is to determine whether or not the performance of the overlap determination stage of an OLC-based assembler (e.g. MIRA) can be improved by using a GPU-based implementation of the Smith-Waterman algorithm, given the current state of GPU technology (“performance” refers to increased alignment sensitivity and reduced completion time). By taking advantage of the highly parallel many-core<sup>1</sup> architecture of GPUs, this thesis aims to incorporate the most sensitive method of identifying overlaps between two sequences into an assembly software (MIRA) while significantly reducing its runtime on commonly available and inexpensive hardware.

### 2.1 Research goals

In order to resolve the question of whether or not it is possible to improve the performance of OLC-based assemblers by using GPU-based acceleration techniques, the following research goals must be achieved:

1. Determining a set of GPU-accelerated alignment software that can be used to accurately find potential overlaps between each pair of sequences from a given list in a timely manner.
2. Exploring the techniques used in the chosen GPU-accelerated alignment software (step 1).
3. Determine methods to reduce the computing time of the chosen software. Adapt and improve the techniques used in the chosen GPU-accelerated alignment software and evaluate the accuracy of the result.
4. Compare the performance of the resultant software with the standard implementation of the sequential Smith-Waterman algorithm and the technique used by MIRA assembler (i.e. adapted k-band Smith-Waterman). Incorporate the new technique into MIRA, should it provide better performance (i.e. reduced completion time and increased sensitivity) guarantees for nucleotide sequence reads alignment compared to the existing approach.

This study involves enhancement and alteration of an existing GPU-accelerated package in order to align medium-length reads sampled from highly complex environments with very small sequence length deviation.

---

<sup>1</sup>Many-core typically refers to devices with dozens or hundreds of cores. Core is considered to be a single computing component responsible for reading and executing program instructions.

## 2.2 Limitations

This thesis work is subject to the following limitations:

1. The work presented within this thesis focuses on the alignment step of the assembly process.  
Although a brief overview of existing assembly strategies is given, detailed exploration or comparison of assembly algorithms are not subject of this work.
2. This thesis does not include an in-depth examination of MIRA assembly software. Instead, the focus of the thesis work is to obtain an adequate understating of the overlap determination stage of this software tool in order to determine methods to improve the performance of this stage.
3. This thesis does not include a thorough comparison of GPU-accelerated sequence alignment software. As part of this work, the published descriptions of several software are compared to determine their suitability for the purpose of this study. Although this work includes a brief explanation of all the software that contributed to this research, it does not contain a comparative evaluation of software and techniques which were examined and rejected.

# CHAPTER 3

## BACKGROUND

This chapter describes material necessary to understand the content of the remainder of this thesis.

Section 3.1 defines sequence assembly and briefly summarizes the most important assembly strategies existing as of this writing. It then argues for a choice of a most suitable approach for assembling medium-length metagenomic reads considering the results of our preliminary study (described in detail in Chapter 4). It addresses the weak points in the existing strategies and at the end gives an introduction to MIRA's overlap screening and sequence alignment process.

A basic introduction to sequence alignment is given in Section 3.2. Section 3.2.1 discusses the principles of sequence alignment. Section 3.2.2 contains a brief discussion about the ways of quantifying the quality of an alignment and the similarity between two sequences. Section 3.2.3 gives an introduction to the different types of dynamic programming algorithms. Different types of alignment for different circumstances are reviewed in Section 3.2.4. A short survey of heuristic alignment algorithms can be found in Section 3.2.5.

Throughout Section 3.3, the increasingly important role of parallel computing will be discussed. A brief history of graphic processing units, as well as a discussion of the usefulness and the limitations of CPU- and GPU-based computing techniques will be given in this section.

Section 3.4 contains a more detailed discussion about the CUDA parallel computing platform and programming model. Section 3.4.1 gives a brief introduction of the CUDA architecture. This architecture is discussed in more detail in Section 3.4.2 and the differences between CUDA's Fermi and Kepler architectures are reviewed in Section 3.4.3.

It must be noted that this chapter cannot be an exhaustive treatment of the topics covered. Instead brief summaries of the most important aspects of the topics are provided. Ideas have been simplified for the context of this study. For more advanced information about sequence alignment please refer to Chapters 4, 5, and 6 of the text by Zvelebil and Baum [20]. For more information about the CUDA architecture and its programming model please see the text by Kandrot and Sanders [21].

### 3.1 Sequence assembly

Once sequencing reads have been produced, it is often necessary to merge the fragments in order to reconstruct the original sequence. This process is known as sequence assembly. A sequence assemblage can be defined as

a hierarchical data structure that maps the sequence data to a putative reconstruction of the target sequence. It combines reads into contiguous sequences (or contigs) and contigs into scaffolds. Contigs provide a multiple sequence alignment of reads plus the consensus sequence.

### 3.1.1 Brief overview of existing assembly strategies

Software tools that are currently used for *de novo* assembly can be classified into two broad categories: string-based assemblers and graph-based assemblers. String-based assemblers, which are implemented with a greedy-extension algorithm<sup>1</sup>, are mainly employed for the assembly of small genomes, while the graph-based assemblers, whether using overlap-layout-consensus or de Bruijn graph methods, are designed to handle complex genomes [17]. Assemblers that use the Eulerian path approach are commonly utilized for short sequences ( $\sim 75 - \sim 150$ bps) whereas those based on the classical overlap-layout consensus paradigm usually handle longer reads ( $\sim 150$ bps and longer).

Over the years assemblers that use the de Bruijn graph approach have proven to resolve the “repeat problem”<sup>2</sup> in assembly and produce relatively accurate solutions for large-scale sequencing problems in a timely manner [14]. Because of the prevalence of the new generation of short-read sequencing technologies (i.e. Illumina) this approach has become the dominant solution. However the future of this technique is not clear considering the progress of new sequencing technologies with longer and more inaccurate sequences as de Bruijn graph assemblers have not been widely used for assembly of longer sequences [22].

Concerns about the complexity of overlap computation have limited the application of the overlap-layout-consensus (OLC) approach and decreased the popularity of it. Despite this the OLC concept remains a point of interest. As our preliminary results show (Chapter 4), OLC-based assemblers are still the best choice for assembling medium-length metagenomic reads. According to our results MIRA, although slow, slightly outperformed the other assembly software tools. It scored very well in size statistics and correctness analysis while producing a very large aggregated contig size. The second-best assembler was Newbler, also an OLC-based assembler, producing the longest contigs and the largest N50 and N80 values as well as very good correctness scores (see Section 4.2).

### 3.1.2 Overlap-layout-consensus strategy

Assembly software using the OLC approach relies on the basic premise that two sequence reads originating from the same place in the genome share a common subsequence (overlap). Using such overlaps between the sequences, an assembler can represent the relationships between the reads as a graph, where nodes represent the reads and an edge connects two nodes if the corresponding reads overlap. The assembly problem thus

---

<sup>1</sup>This thesis assumes that the readers are familiar with the basic principals of sequence assembly as well as general computational assembly techniques (e.g. greedy-extension, de Bruijn, and Eulerian path algorithms). Readers not familiar with these are referred to the text by Scheibye-Alsing et al. [22].

<sup>2</sup>Identical and nearly identical sequences (known as repeats) can increase the time and space complexity of assembly algorithms exponentially.

becomes the problem of identifying a Hamiltonian path (i.e. a path in an undirected or directed graph that visits each node exactly once) which is NP-complete [12].

As established above, the fundamental phase of assembly software using the overlap-layout-consensus approach is, as the name suggests, computing the overlaps. The most accurate methods for detecting overlaps between sequences, even partial ones, are the dynamic programming algorithms for global alignments introduced by Needleman and Wunsch in 1970 [23] and later refined for local alignments by Smith and Waterman in 1981 [24]. The run-time complexity of these algorithms is  $O(n \times m)$ , with  $n$  and  $m$  being the lengths of the two sequences. Considering the number of comparisons which have to be completed to detect overlaps between all the sequences in a dataset (in this situation,  $n$  and  $m$  are each the total combined length of all the reads residing in a dataset), these algorithms are unacceptable for most screening procedures. These inefficient methods represent the most important bottleneck in the process of assembly using the OLC approach. This is the reason why almost all OLC-based assembly software available today employs some faster, heuristic, string searching method often originated from the word-based method introduced by Wilbur and Lipman [25].

Being heuristics, the word-based methods only offer an approximate solution that does not always find all the potential overlaps between two sequences. A key point in this requirement is the ability to find – besides long overlaps with low error rates expected to be recognized by any algorithm – even weak overlaps and overlaps in regions with high error rates. Weak overlaps are characterized either by only a small number of bases from each sequence overlapping each other or by overlapping bases having a high error rate, or both.

Of all the assembly software tools proposed up to now, MIRA is the only one that uses a modified version of the Smith-Waterman dynamic programming algorithm for examining potential overlaps between two sequences. This could very well explain MIRA’s good size statistics and correctness scores as well as its poor performance.

### 3.1.3 MIRA: an automated genome and EST assembler

Taking into account the relatively good results of the MIRA assembly program as well as its unsatisfactory speed, modifying this software in order to achieve a faster and more scalable assembler is desirable. The new version of the program should be able to more efficiently tackle the sequence alignment step of the assembly process in order to facilitate the processing of large and complex metagenomic data.

A closer look at MIRA’s overlap screening and sequence alignment step reveals that after performing different refinement steps to the original read data (e.g. sequencing vector clipping, standard repeat tagging, quality clipping<sup>3</sup>, etc.), MIRA finds the high confidence region (HCR) for each read and compares it to the HCR of every other read to see if they could match and have overlapping parts. Performing this process in a timely manner is only possible by taking advantage of very quick heuristic filters employed by MIRA (i.e.

---

<sup>3</sup>This thesis assumes that the readers are familiar with the concepts of sequencing-vector clipping, standard repeat tagging, and quality clipping. Readers not familiar with these are referred to the text by Chervaux [12].

ZEBRA and DNASAND [12].). All the possible overlaps then form an initial building graph.

DNASAND and ZEBRA not only provide information on whether two sequences are similar enough to warrant a sequence alignment, but they also provide the approximate offset for the sequence alignment (more discussion on this topic is presented in Section 3.2). Using this prior knowledge and assumptions about the alignment of the sequences, the reads in the initial building graph (as mentioned before, an assembler can represent the relationships between the reads as a graph) which could have overlaps are reviewed with an adapted version of the Smith-Waterman algorithm. Obvious mismatches are rejected and removed from the initial building graph. The accepted read-pairs are inserted into one or several alignment graphs. These alignment graphs define all the assemblies that are possible with the given reads [12].

## 3.2 Sequence alignment

Sequence alignment is the most common task performed by bioinformaticians. Procedures relying on sequence comparison are diverse and range from database searches [26] to secondary structure prediction [27]. Sequences can be compared pairwise when screening databases for similar sequences, or they can be multiply aligned to visualize the effect of evolution across a whole protein family. In this section the theory underlying sequence alignment will be discussed, as well as different methods of quantifying sequence similarity, different types of alignments and various computational approaches to the sequence alignment problem.

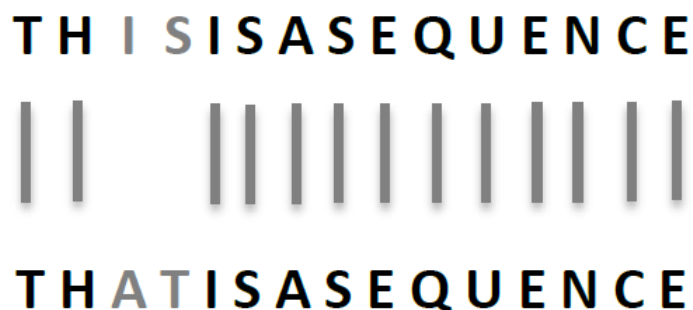
### 3.2.1 Principles of sequence alignment

The way of arranging the sequences of DNA, RNA, or protein in order to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences is called sequence alignment. Ideally what is expected to be achieved when comparing sequences is that they can be lined up in such a way that identical or similar bases or amino acids are matched with each other to the maximum possible extent.

To represent a protein or nucleotide sequence, a symbolic sequence can be used. In a symbolic sequence each base or residue in each sequence is represented by a letter (i.e. Adenine (A), Cytosine (C), Guanine (G) and Thymine (T) are used for textual representation of DNA). The convention is to print the single-letter codes for the constituent monomers in order (from 5' to 3' of a sequence). This is based on the assumption that the combined monomers are evenly spaced along the single dimension of the molecule's primary structure. Figure 3.1 illustrates the general principle.

As illustrated in Figure 3.1, with short and similar sequences, an alignment can clearly identify the similarities between sequences. However, when sequences become more different from each other, it becomes more difficult to compare them. How should two sequences in which mutation has led to insertion or deletion of one or more residues be aligned? To get around this problem, gaps are introduced into one or both of the sequences so that maximum similarity is preserved. When a residue in one sequence seems to have been

**Figure 3.1:** The general principle of sequence alignment. The characters in black are identical.

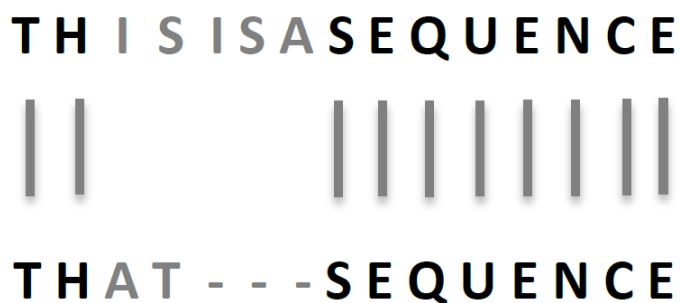


THIS IS A SEQUENCE

THAT IS A SEQUENCE

deleted, the residue’s “absence” is labelled by a dash (or “gap”) in the other sequence. When a residue appears to have been inserted to produce a longer sequence, a dash appears opposite in the un-augmented sequence. The action of inserting such spaces is known as gapping. Although completely separate biological events, when it comes to aligning two sequences, a deletion in one sequence is symmetric with an insertion in the other. That is when sequences  $a$  and  $b$  are gapped relative to another, a deletion in sequence  $a$  can be seen as an insertion in sequence  $b$  (Figure 3.2). Indeed, the two types of mutation are referred to together as indels. It must be borne in mind that although use of gaps to achieve a similar match is necessary in many cases, they must be used carefully (i.e. using a large number of gaps can lead to meaningless alignment).

**Figure 3.2:** Gapping preserves the maximum similarity between two sequences. In this figure gapping is used to preserve the similarity between the sequences used in Figure 3.1. A mutation has led to insertion of 3 residues (i.e. I, S and A) on one of the original sequences.

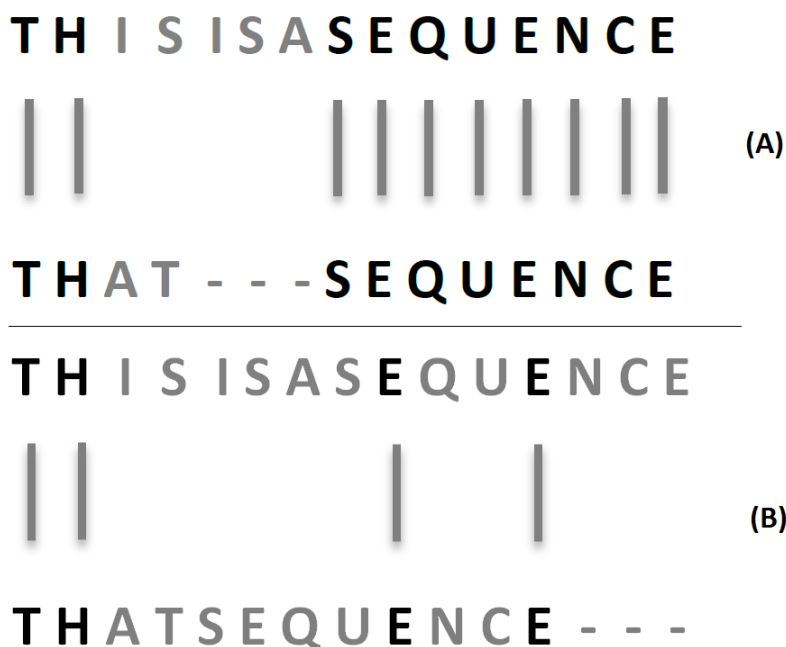


THIS IS A SEQUENCE

THAT - - - SEQUENCE

There is never just one possible sequence alignment between any two sequences (Figure 3.3) and the most similar alignment is often not obvious. Hence, at the heart of all the sequence-alignment techniques, an algorithm exists to test the similarity of many of the generated alignments, giving it a score and filtering out the unsatisfactory results [20].

**Figure 3.3:** There is never just one possible sequence alignment between any two sequences. Cases (A) and (B) use gapping in different ways to preserve the similarity between the sequences. As apparent below, alignment (A) is the “most similar alignment”.



### 3.2.2 Scoring alignments and substitution matrices

Because it is possible for two sequences to be aligned in a variety of different ways (Figure 3.3), a ranking technique is needed to objectively determine the best possible alignment for any given pair of sequences, given some set of scores associated with aligning the various letters. To this end, a numerical value or a score for overall similarity of each possible alignment is needed. Given that there are many algorithmic ways to solve optimization problems, many alignment methods are able to find the best alignment between two strings under some scoring scheme. These scoring schemes can be as simple as percentage identity between two sequences or a ‘reward for a match, penalty for a mismatch’ technique. Indeed, many early sequence alignment algorithms were described in these terms.

Nucleotide similarity matrices are used to align nucleic acid sequences and they tend to be much simpler than protein similarity matrices. For example, a simple matrix will assign identical bases a score of +1 and non-identical bases a score of -1. A more complicated matrix would give a higher score to transitions (changes from a pyrimidine such as C or T to another pyrimidine, or from a purine such as A or G to another purine) than to transversions (from a pyrimidine to a purine or vice versa).

However to find a scoring scheme capable of identifying the most similar alignment, it is important to take into account the following:



- Biological molecules have evolutionary histories, three-dimensional folded structures, and other features which constrain their primary sequence evolution [28].
- Due to indels (i.e. insertions and deletions) that have occurred during the course of evolution, homologous sequences are often of different lengths. Use of gaps to achieve as similar a match as possible is then necessary.

In addition to the mechanics of alignment and comparison, the scoring system itself requires careful thought and can be very complex. One of the first steps towards developing a sensitive scoring scheme was the introduction of probabilistic matrices for scoring pairwise amino acid alignments first developed by Margaret Dayhoff and her co-workers in the 1960s and 1970s [29]. These serve to quantify evolutionary preferences for certain substitutions over others. More probabilistic modelling approaches (e.g. BLOSUM, STR, SLIM, PHAT, etc.) have been brought gradually into computational biology by many routes [28]. As of the date of this study, there are 94 different scoring matrices collected in a list called AAINDEX [30], each designed to work well in special situations. With many scoring matrices available today the choice of substitution matrix essentially depends on the problem to be solved. As a general rule, one must always take into account the degree of evolutionary distance and the length of the sequences when choosing a suitable scoring matrix [20].

### 3.2.3 Dynamic programming algorithms

Having discussed scoring schemes, we want to incorporate these techniques to find optimal alignments. The most obvious way to find the best (most similar) alignment with gaps would be to generate all possible gapped alignments, find the score for each and select the highest-scoring alignment (i.e. brute force enumeration). This is impractical because the number of possible alignments becomes tremendously large even for short sequences. Therefore dynamic programming approach must be utilized. Dynamic programming is a method by which a larger problem may be solved by first solving smaller, partial versions of the same problem. The dynamic programming approach seeks to solve each subproblem only once, thus reducing the number of computations: once the solution to a given subproblem has been computed, it is stored and the next time the same solution is needed, it is simply looked up. This approach is especially useful when the number of repeating subproblems grows exponentially as a function of the size of the input which is the case of gapped alignments.

### 3.2.4 Types of alignment

There are two main types of alignments: global and local.

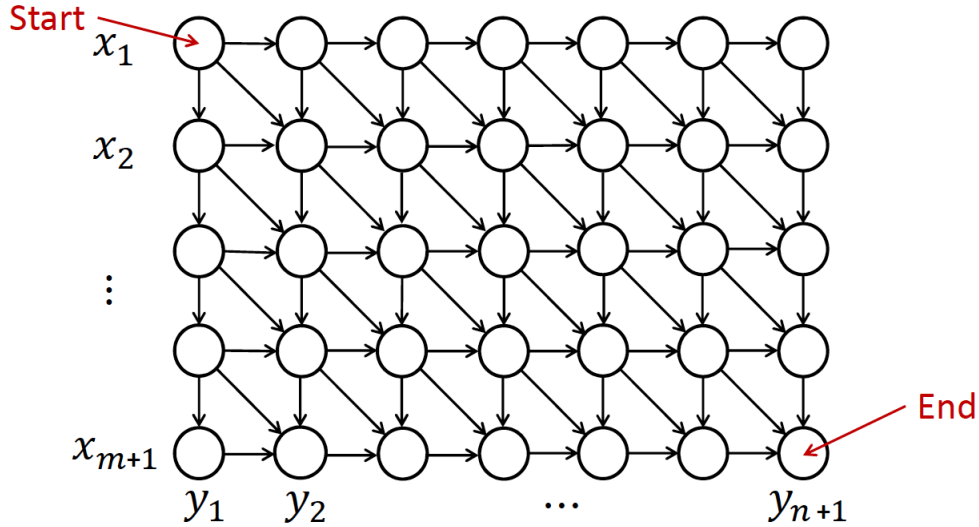
**global alignment:** find the best alignment of one entire sequence with another entire sequence [31].

**local alignment:** find the best alignment of any segment of one sequence against any segment of another sequence [31].

The key concept in all these algorithms is to calculate a matrix  $S$  of optimal scores of sub-sequence alignments. The matrix has  $(m + 1)$  rows and  $(n + 1)$  columns. The rows correspond to the residues of one of the sequences and the columns to those of the other sequence ( $x$  and  $y$  in Figure 3.4).

A widely used global alignment algorithm is Needleman-Wunsch algorithm. A global alignment such as one produced by this algorithm covers the full range of each sequence and produces the optimal alignment. This algorithm, as the name suggests, was first described by Needleman and Wunsch [23] and later modified by Sellers [32], Gotoh [33], and others [34].

**Figure 3.4:** A global alignment may be viewed as a path through a directed path graph. Figure is taken from Altschul et al. [35].



A global alignment may be viewed as a path through a directed graph which begins at the upper left corner of the scoring matrix (matrix  $S$ ) and ends at the lower right (Figure 3.4). Diagonal steps correspond to matches or substitutions, while horizontal or vertical steps correspond to indels. The increments or decrements of scores are associated with each edge, and scores for alignments are associated with the nodes. The score at a node is the score of the best alignment “ending” at that node. For instance, the score at the node  $x_n$  and  $y_n$  is the score of the best alignment starting from  $x_0$  and  $y_0$ , and ending at node  $x_n$  and  $y_n$ . Hence, each alignment corresponds to a unique path, and vice versa. Traceback information can be calculated, starting with the score at the bottom, right node. Traceback then starts at the final node. In this algorithm it is efficient to record traceback information to identify which edge or edges led to the optimal score at each node [26].

Note that there may be more than one optimal alignment if at some point along the path during traceback an element is encountered that was derived from more than one of the three possible alternatives coming into each node. This algorithm does not distinguish between these possible alignments, although there may

be reasons for preferring one to another. Such preference would normally be justified with further knowledge about the problem at hand.

This approach to global sequence alignment can briefly be explained in three steps:

1. **Setting up a matrix:** First the values of the top row and leftmost column are initialized. In the initialization phase, the score of each cell is set to the gap score multiplied by the distance from the origin. Gaps may be present at the beginning of either sequence, and their cost is the same as anywhere else.
2. **Scoring the Matrix:** After setting up the matrix, at each step a choice is forced among match/mismatch, insertion or deletion, even if none of these choices are favorable (i.e. if the choice will degrade the score along a path containing a well-fitting local region) [31]. Any matrix element  $S_{i,j}$ , can be filled using the following steps:

$$S_{i,j} = \max \begin{cases} S_{i-1,j-1} + s(x_i, y_j) & x_i \text{ and } y_j \text{ aligned} \\ S_{i-1,j} + g & x_i \text{ aligned with a gap} \\ S_{i,j-1} + g & y_j \text{ aligned with a gap} \end{cases} \quad 1 < i \leq m, \quad 1 < j \leq n \quad (3.1)$$

where  $W_i$  is the gap-scoring scheme and  $s(x_i, y_j)$  is a similarity function (substitution matrix).

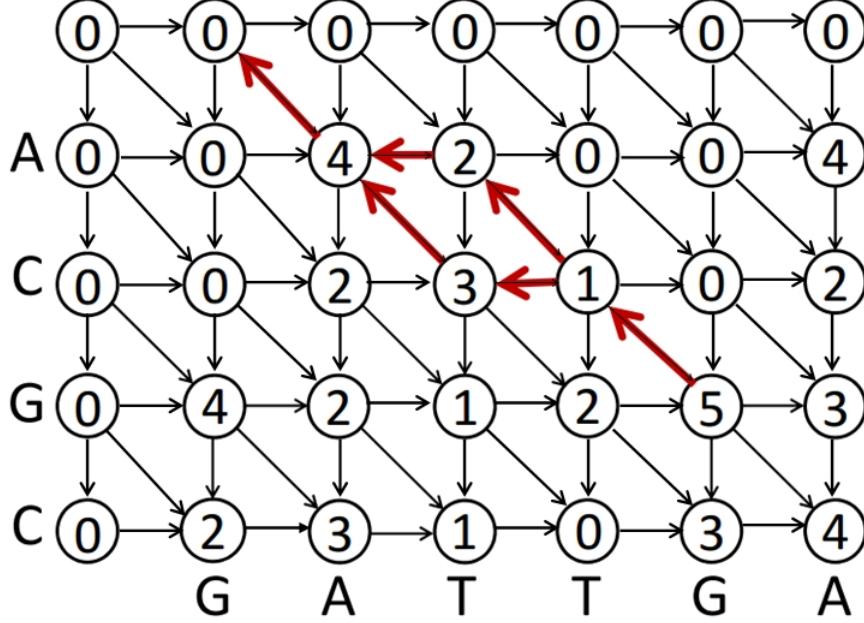
3. **Identifying the optimal alignment:** For global alignment, traceback starting at the lower-right cell to determine the actual alignment (see Figure 3.4).

To analyze the time complexity of the Needleman-Wunsch algorithm, we can essentially analyze each individual part of the algorithm. To initialize the matrix, we need to input the scores of the row 0 and column 0. This has a time complexity of  $O(m + n)$  ( $m$  and  $n$  are the length of the sequences  $x$  and  $y$  respectively). The next step is filling in the matrix with all the scores,  $F(i, j)$ . As discussed before, for each cell of the matrix, three neighboring cells must be compared, which is a constant time operation. Thus, to fill the entire matrix the time complexity is the number of entries, or  $O(mn)$ . Finally the traceback requires a number of steps. The first step is marking the cells according to the rules above, the complexity of  $O(m + n)$ . The second step is finding the final path which involves jumping from cells of matching residues. Since this step can include a maximum of  $n$  cells (where  $n \geq m$ ), this step is  $O(n)$ . Thus, the overall time complexity of this algorithm is

$$O(m + n) + O(mn) + O(m + n) + O(n) = O(mn) = O(n^2).$$

There are many cases where only parts of sequences are similar. In this case, the local alignment approach can produce more useful results. A local alignment such as the method of Smith and Waterman [24] focuses on the regions with the most similarity between two sequences. Smith-Waterman, which is a modification of Needleman-Wunsch algorithm, is the most rigorous method by which substrings of two sequences can be

**Figure 3.5:** An optimal alignment for two sequences using local alignment (i.e. Smith-Waterman) technique with +4, -1, and -2 for match, mismatch, and gap, respectively. Black arrows indicate possible choices between match/mismatch, insertion or deletion. Red arrows illustrate the traceback that is used to determine the actual alignment. Figure is taken from Altschul et al. [35].



aligned. This method of aligning is most useful when searching through a sequence dataset with a query sequence from an unknown source [20, 34, 31].

The basic steps of the Smith-Waterman algorithm are similar to those of the Needleman-Wunsch algorithm. The key difference in the local alignment algorithm from the global algorithm is that whenever the score of the optimal sub-sequence alignment would be less than zero, it is dismissed and the sub-alignment of null string aligned with null string is chosen instead. Another algorithmic difference is that the Smith-Waterman algorithm starts the traceback from the highest-scoring element wherever it occurs (see Figure 3.5).

The basic steps for the Smith-Waterman algorithm are as follow:

1. **Initialization of a matrix:** In the first step of this algorithm, the two sequences are arranged in a matrix form. The values in the first row and the first column are set to zero.

$$S_{i,0} = 0 \quad 0 \leq i \leq m \quad (3.2)$$

$$S_{0,j} = 0 \quad 0 \leq j \leq n \quad (3.3)$$

2. **Filling the matrix with appropriate scores:** The second and crucial step of the algorithm is filling the entire matrix. Therefore it is important to know the neighbor values (diagonal, upper and

left) of the current cell to fill each and every cell. It is worth noting that there is no chance to see any negative values in the matrix, since zero is acceptable as the lowest value. Any matrix element  $S_{i,j}$  can be filled using the following methodology:

$$S_{i,j} = \max \begin{cases} S_{i-1,j-1} + s(x_i, y_j) & x_i \text{ and } y_j \text{ aligned} \\ S_{i-1,j} + g & x_i \text{ aligned with a gap} \\ S_{i,j-1} + g & y_j \text{ aligned with a gap} \\ 0 & \end{cases} \quad 1 < i \leq m, \quad 1 < j \leq n \quad (3.4)$$

where  $W_i$  is the gap-scoring scheme and  $s(x_i, y_j)$  is a similarity function (substitution matrix).

3. **Tracing back the paths for the optimal alignment:** The final step is back tracing. Prior to this step, the highest-scoring element in the entire matrix must be obtained. The traceback begins from the position of the highest-scoring element (wherever it occurs, e.g.  $(p, q)$ ) and continues to one of positions  $(i-1, j)$ ,  $(i, j-1)$ , and  $(i-1, j-1)$  depending on the direction of movement used to construct the matrix. The traceback continues until a matrix cell with a zero value is reached. Once finished, the alignment is reconstructed as follows: Starting with the last value (i.e. the cell at which the back trace ended), reach  $(p, q)$ , the cell with the highest value, using the previously calculated path. A diagonal jump implies there is an alignment (either a match or a mismatch).

It would at first appear that the problem of finding an optimal local alignment should be significantly more complex than the problem of finding an optimal global alignment, because the start and stop position of the alignment must be located as well. However, the additional calculation just adds a constant factor to the complexity of the algorithm. The factor affecting the complexity is that Equation 3.4 has to consider four possibilities, instead of 3 (Equation 3.1). Although this means that a constant of 3 (in the complexity formula) might be increased to 4. The complexity formula itself remains  $O(n^2)$ .

Pairwise sequence alignment is used to identify regions of similarity that may indicate functional, structural and/or evolutionary relationships between two biological sequences. By contrast, multiple sequence alignment is generally the alignment of three or more biological sequences (protein or nucleic acid) that assists researchers to better infer homology and study the evolutionary relationships between the sequences.

Multiple sequence alignment is an extension of pairwise alignment to incorporate more than two sequences at a time. Multiple alignment methods try to align all of the sequences in a given set. Multiple sequence alignments are computationally difficult to produce and most formulations of the problem lead to NP-complete combinatorial optimization problems [36].

While the transition from pairwise to multiple sequence alignment is conceptually straightforward, the obvious algorithm to compute the exact solution takes an amount of time exponential in  $k$  (i.e. number of sequences to be aligned). It is therefore almost impractical to use this method in real world problems.

Nevertheless, the utility of these alignments in bioinformatics has led to the development of a variety of methods suitable for aligning three or more sequences. For more information on this topic please refer to Simossis et al. [37].

### 3.2.5 Algorithmic approximations

Despite the development of new computing technologies which has led to more powerful computers, the ever-growing size of sequence databases and their daily updating makes the above-described full-matrix dynamic programming techniques far too demanding for general use in database searches (where  $m$  and  $n$  are very big). Alternative approaches, hence, were created.

A number of alternative approaches have been developed (e.g. BLAST [26], and FASTA [38], etc) that are considerably faster. The key to the success of these methods is the use of indexing techniques to locate possible high-scoring short local alignments. These initial local alignments are then extended, subject to certain constraints, to provide scores that are used to rank high-scoring pairs based on their similarity. In most implementations, a modified dynamic programming algorithm is used to examine the best-scoring alignments and produce final scores and alignment [20]. The speedup, however, comes at a price: the heuristic methods cannot guarantee to find the highest-scoring alignment.

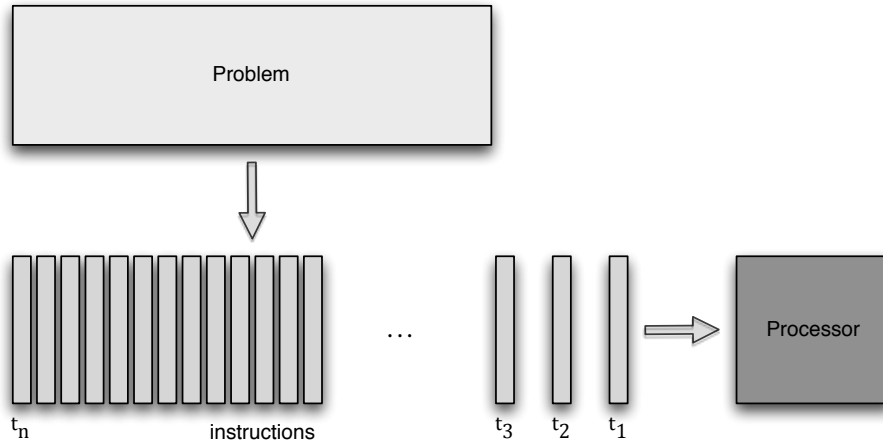
The work presented within this thesis focuses on the dynamic programming approach to the sequence alignment problem. Thus, detailed descriptions of heuristic alternatives of these techniques are not subject of this work. For more information about these methods employed for sequence alignment please refer to Hirosawa et al. [39].

### 3.3 Parallel computing

Traditionally, most software has been written for serial computation to be run on a single computer having a single Central Processing Unit (CPU). With this approach, a problem is broken into a discrete series of instructions that are executed one after another. One and only one instruction may execute at any moment in time (Figure 3.6).

Fast runtime has always been a necessity and a competitive advantage in software development. Accordingly, there has always been the pressure to make applications run faster. Historically, as processors have increased their speed, the needed speedups could often be achieved by tuning the single CPU performance of the program and by utilizing the latest and fastest hardware. However, since the power consumed by the fastest possible processors generates too much heat when in use, we will no longer see significant increases in the clock speed of processors. So what should be done when the problem is too costly to be solved with a classical approach?

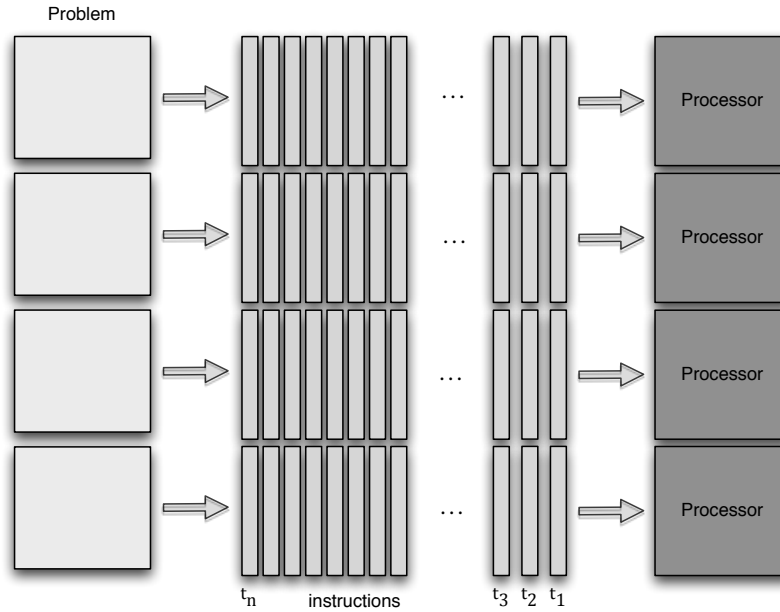
**Figure 3.6:** Serial computation. In this approach the problem is broken into a discrete series of instructions. These instructions are executed one after another by the processor. Taken from the text by Barney [40].



One way to solve the problem is to break the problem into pieces, and arrange for all the pieces to be solved simultaneously. This approach is referred to as parallel computing (Figure 3.7). In the simplest sense, parallel computing is the simultaneous use of multiple compute resources (whether CPU or GPU) to solve a computational problem. The more pieces, the faster the job reaches a point where the pieces become so small that the process of breaking-up and distributing the computation, and consequently combining the result dominates the execution time and there is no more gain in efficiency. [41].

Parallel computing is not applicable to all problems. The computational problem should be able to be broken apart into discrete pieces of work that can be solved simultaneously. The pieces that cannot be parallelized will limit the overall speedup available from parallelization. This puts an upper limit on the

**Figure 3.7:** Parallel computing. In this approach, each part is further broken down to a series of instructions which execute simultaneously on different processors by employing a suitable control/co-ordination mechanism. Taken from the text by Barney [40].



usefulness of parallel execution.

Parallel computing can take many forms sharing the same concept. Compute resources can be CPUs being inter-connected using any of a large number of schemes or Graphics Processing Units (GPUs), or a combination of both. The number of processors can range from a few to thousands. These processors on which a parallel program executes may all be packed into one box (yielding a “multiprocessor” or “parallel compute”) or they may be separate, autonomous machines connected by a network. The network might be local or wide-area; the computers in the network might be any architecture [41].

There are different ways to classify parallel computers. One of the most widely used classifications is known as Flynn’s Taxonomy [42]. Flynn’s taxonomy distinguishes multi-processor computer architectures according to two independent dimensions of instruction stream and data stream. Each of these dimensions can have only one of two possible states: single or multiple [40]. Four possible classifications according to Flynn are:

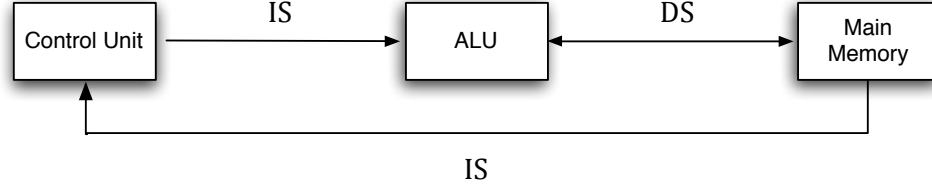
**Single Instruction, Single Data (SISD):** In this organization, sequential execution of instructions is performed by one processor containing a single processing element (PE), (e.g. ALU<sup>4</sup> under one control unit as shown in Figure 3.8). SISD machines are conventional serial computers that process only one stream of instructions and one stream of data [40]. This corresponds to the Von Neumann

<sup>4</sup>An arithmetic logic unit (ALU) is a digital circuit that performs integer arithmetic and logical operations. The processors found inside modern CPUs and GPUs accommodate very powerful and very complex ALUs.



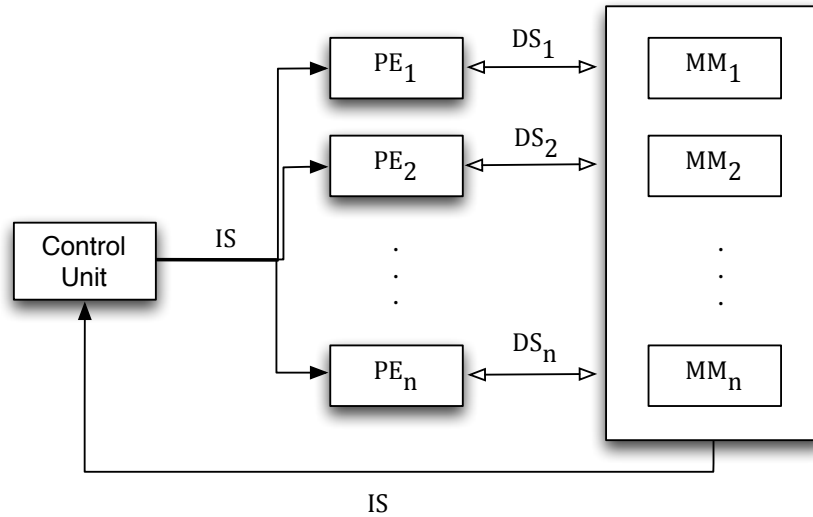
architecture<sup>5</sup>. For more information, please refer to the text by Von Neumann [43].

**Figure 3.8:** Single Instruction, Single Data (SISD) organization. *IS* and *DS* represent the single instruction stream and single data stream, respectively. Taken from the text by Barney [40].



**Single Instruction, Multiple Data (SIMD):** In this organization, multiple processing elements work under the control of a single control unit (CU). It has one instruction stream and multiple data streams. All the processing elements of this organization receive the same instruction from the CU. Main memory can also (but not necessarily) be divided into modules for generating multiple data streams acting as a distributed memory <sup>6</sup> as shown in Figure 3.9. It's notable that most modern computers, particularly those with graphics processor units (GPUs), are designed based on the SIMD method [40]. This architecture corresponds to the Modified Harvard architecture [44].

**Figure 3.9:** Single Instruction, Multiple Data (SIMD) organization. *IS* and *DS* represent a single instruction stream and various data streams, respectively. Taken from the text by Barney [40].



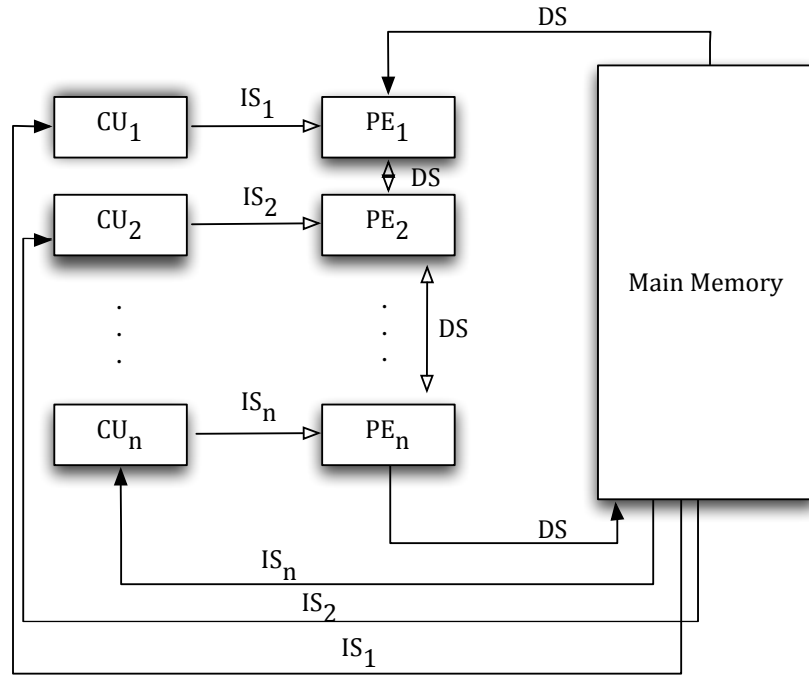
**Multiple Instruction, Single Data (MISD):** In this organization, multiple processing elements are organized under the control of multiple control units. Each control unit (CU) is handling one

<sup>5</sup>A design architecture for an electronic digital computer with subdivisions of a processing unit, a control unit, a memory to store both data and instructions, external mass storage, and input and output mechanisms.

<sup>6</sup>Distributed memory refers to a multiple-processor computer system in which each processor has its own private memory

instruction stream (IS) through its corresponding PE. But each PE is processing only a single, common data stream (DS) at a time. Therefore, for handling multiple instruction streams and a single data stream, multiple control units and multiple processing elements are required. In this approach all processing elements are interacting with the common shared memory for the organization of a single data stream as shown in Figure 3.10 [40].

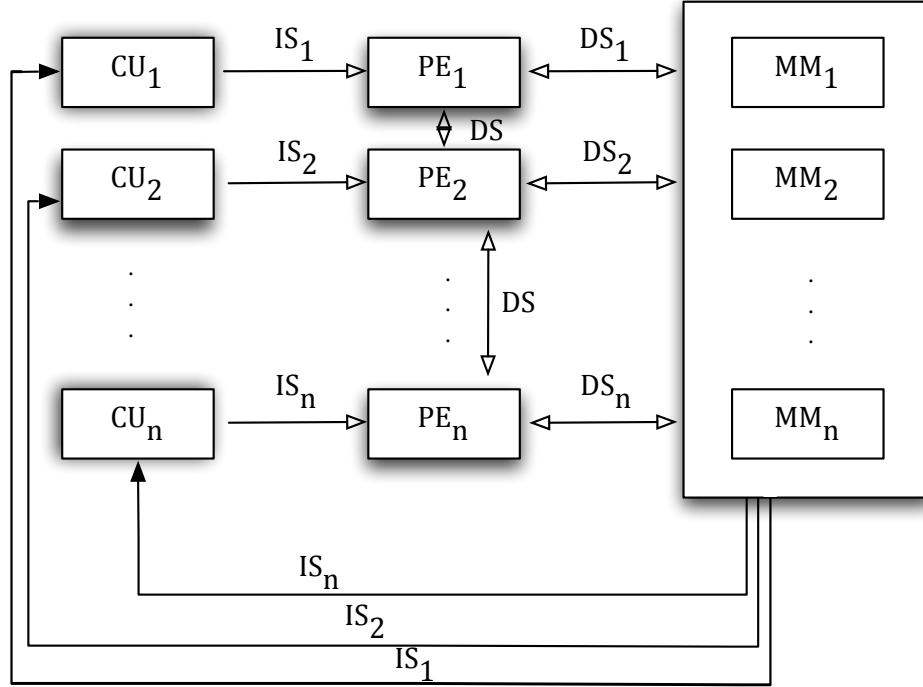
**Figure 3.10:** Multiple Instruction, Single Data (MISD) organization. *IS* and *DS* represent number of instructions stream and single data stream, respectively. Taken from the text by Barney [40].



**Multiple Instruction, Multiple Data (MIMD):** In this organization, multiple processing elements (PE) and multiple control units (CU) are organized as in MISD. However, in this organization multiple instruction streams operate on multiple data streams (DS). Therefore, for handling multiple instruction streams, multiple control units and multiple processing elements are organized such that multiple processing elements are each handling a separate data stream from the main memory, as shown in Figure 3.11 [40].

Historically, parallel computing has been considered to be “the high end of computing”, and programs using this approach have been used to model difficult problems in many areas of science and engineering, including bioscience, biotechnology, and genetics. This study mainly focuses on the SIMD parallel computing model for current-generation parallel machines utilizing GPUs as an alternative to the sophisticated parallel tools commonly used for speeding-up the dynamic programming approaches to the sequence alignment problem.

**Figure 3.11:** Multiple Instruction, Multiple Data (MIMD) organization.  $IS$  and  $DS$  represent multiple instructions and data streams, respectively. Taken from the text by Barney [40].



### 3.3.1 CPU vs. GPU

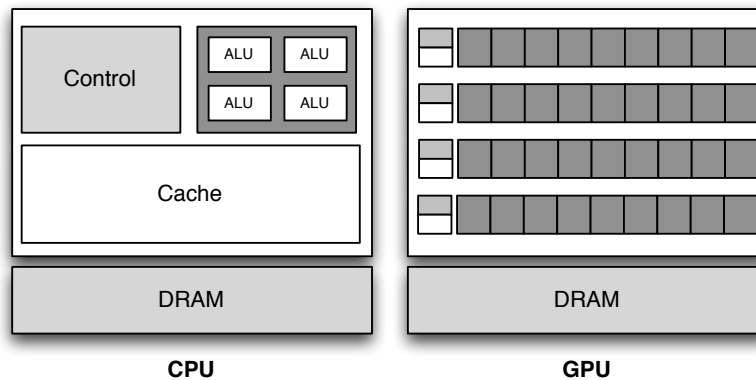
As discussed earlier, the complexity of dynamic programming techniques commonly used for computing optimal pairwise sequence alignments is quadratic with respect to the lengths of alignment targets, which makes these techniques time consuming for applications involving large datasets. Therefore heuristic methods have been introduced as an alternative approach. The drawback is that the more efficient the heuristics, the worse the quality of the result. Hence, use of high-performance computing techniques is considered a good alternative for getting high-quality results in a timely manner.

Two major computing platforms are deemed suitable for this purpose. The first one is the multi-core and multi-threaded CPUs that are capable of running many types of applications. As discussed above, because of various fundamental limitations in the fabrication of integrated circuits, it is no longer feasible to rely on ever-increasing processor clock speeds as a means for extracting additional power from existing architectures. Because of power and heat restrictions as well as a rapidly approaching physical limits to transistor size, in 2005 leading CPU manufacturers began offering processors with two computing cores instead of one. This development has been followed with the release of three-, four-, six-, eight-, and twelve-core central processor units over the last few years [21].

The second one is the GPU that is designed for graphics processing with many small processing elements. The massive processing capability of GPUs gave rise to the general purpose-GPU field [45] [46]. Fundamentally, CPUs and GPUs are built based on very different philosophies. CPUs are designed for a wide variety of applications and to provide fast response times to a single task. Architectural advances such as branch prediction, out-of-order execution, and frequency scaling<sup>7</sup> have been responsible for performance improvement. However, these advances come at the price of increasing complexity/area ratio and power consumption. As a result, mainstream CPUs today can pack only a small number of processing cores on the same board in order to stay within the power and thermal constraints. GPUs, on the other hand, are built specifically for rendering and other graphics applications that have a large degree of data parallelism (each pixel on the screen can be processed independently). Graphics applications are also latency tolerant. As a result, GPUs can trade off single thread performance for increased parallel processing.

There is little doubt that today's CPUs would provide the best single thread performance for high-throughput computing workloads. However, the limited number of cores in today's CPUs limits how many pieces of data can be processed simultaneously. On the other hand, GPUs provide many parallel processing units which are ideal for high performance computing. However, the design for graphics pipeline lacks some critical processing capabilities (e.g., large caches) for general purpose work loads, which may result in lower architecture efficiency [47]. With that being said, for the right kind of problem, by developing an appropriate algorithm for process streaming and data dependency handling of the modern GPUs, significant speedups can be achieved [48, 49, 50, 51, 52, 53, 54].

**Figure 3.12:** Differences between CPU and GPU architectures. Taken from CUDA programming guide [55].



In a nutshell, the GPU is specialized for compute-intensive, highly data-parallel computation (owing to its graphics rendering origin), and therefore designed such that more transistors are devoted to data processing rather than data caching and control flow as illustrated in Figure 3.12. More specifically, the GPU is especially well-suited to address problems that can be expressed as data-parallel computations (the same

<sup>7</sup>Frequency scaling is the technique of ramping up a processor's frequency so as to achieve performance gains.

program is executed on many data elements in parallel) with high arithmetic intensity (the ratio of arithmetic operations to memory operations). Because the same program is executed for each data element, there is a lower requirement for sophisticated flow control [55]. For further discussion about the differences between the performance and computing characteristics of architectural features on today’s CPUs and GPUs, please refer to Lee et al. [47].

## 3.4 GPU computing

In comparison to the central processor’s traditional data processing pipeline, performing general-purpose computations on a graphics processing unit is a new, yet effective approach. In fact, the GPU itself is relatively new compared to the computing field at large. The following gives a brief history of GPUs.

With the growth in popularity of graphically driven operating systems such as Microsoft Windows in the late 1980s and early 1990s, a new market appeared for a new type of processor. Around the same time, Silicon Graphics, a company popularizing the use of three-dimensional graphics as well as providing the tools to create cinematic effects, released the OpenGL library to be used as a standardized, platform-independent method for writing 3D graphics applications. The mid-1990s saw the release of a new generation of games (i.e. “first-person shooter” genre games such as Doom and Quake) and the growth in their popularity led to competition to create progressively more realistic 3D environments for PC gaming. At this time companies such as NVIDIA, ATI Technologies, and 3dfx Interactive began releasing graphics accelerators that were affordable enough to attract widespread attention [21]. The release of NVIDIA’s GeForce 256 further pushed the capabilities of consumer graphics hardware and enhanced the potential for even more applications. NVIDIA’s release of the GeForce 3 series in 2001 was an important breakthrough in GPU technology. The GeForce 3 series was the computing industry’s first chip to implement Microsoft’s then-new DirectX 8.0 standard, which for the first time gave developers some control over the computations that would be performed on their GPUs. However, because standard graphics APIs<sup>8</sup> such as OpenGL and DirectX were still the only way to interact with a GPU, any attempt to perform arbitrary computations on a GPU were still subject to severe programming and hardware constraints [21, 55].

### 3.4.1 Why CUDA?

The high arithmetic throughput of GPUs attracted many researchers to the possibility of using graphics hardware for more than graphic-processing purposes. However, the general approach in the early days of GPU computing was extremely complex and difficult to follow and, as mentioned above, any attempt to perform generic computations on a GPU were subject to hardware limitations and constraints of programming within a graphics API. In essence, developers had to trick the GPU into performing non-rendering tasks by making those tasks appear as if they were OpenGL or DirectX standard renderings. The resultant programming

---

<sup>8</sup>An application programming interface specifies how some software components should interact with each other.

model was far too restrictive (i.e. there were tight resource constraints on type of input data and memory usage). Also, there existed no good method of debugging for any code that was being executed on the GPU, nor was there any way to terminate a faulty program. Furthermore, programmers who wanted to use a GPU to perform general-purpose computations would need to learn OpenGL or DirectX which were convoluted [21, 55].

In 2006, NVIDIA unveiled the GeForce 8800 GTX, which was the first GPU to be built with NVIDIA’s CUDA (Compute Unified Device Architecture) [21]. This architecture included several components specifically designed for generic GPU computing. CUDA was released along with a software environment that allowed developers to use C as a high-level programming language to develop application software that transparently scales its parallelism to leverage the large number of processor cores available on the GPU while maintaining a low learning curve for programmers familiar with standard C. Table 3.1 illustrates languages, application programming interfaces, or directives-based approaches which are supported by the latest version of CUDA as of writing this thesis (i.e. CUDA 5.5).

**Table 3.1:** GPU computing applications. CUDA supports various languages and application programming interfaces. Taken from *CUDA programming guide* [55].

GPU computing application						
Libraries and Middleware						
CUFFT			VSIPL			
CULBAS	CULA	Thrust	SVM	PhysX	Iray	MATLAB
CURAND	MAGMA	NPP	OpenCurrent	OptX		Mathematica
CUSPARSE						
Programming Languages						
C	C++	Fortran	Java Python Wrappers	DirectCompute	Directives (e.g. OpenACC)	
CUDA-enabled NVIDIA GPUs						
Kepler Architecture	GeForce 600 Series	Quadro Kepler Series	Tesla K20			
			Tesla K10			
Fermi Architecture	GeForce 500 Series	Quadro Fermi Series	Tesla 20 Series			
	GeForce 400 Series					
Tesla Architecture	GeForce 200 Series	Quadro FX Series	Tesla 10 Series			
	GeForce 9 Series	Quadro Plex Series				
	GeForce 8 Series	Quadro NVS Series				

### 3.4.2 CUDA architecture

Since NVIDIA intended its new family of graphics processors to be used for general-purpose computing, their ALUs are built to be in accordance with IEEE requirements for single-precision floating-point arithmetic. They are also designed to use an instruction set particularly developed for general computation rather than specifically for graphics. In this new framework, a unified pipeline allows each and every ALU on the chip to be marshalled by a program intending to perform general-purpose computations [21]. Furthermore, the execution units on the GPU are allowed arbitrary read and write access to memory as well as access to a software-managed cache known as shared memory.

All of these features of the CUDA Architecture were added in order to create a GPU that would be programable for general-purpose computations as well as graphics tasks. The most exciting feature of CUDA, however, was the new language (known with the same name as CUDA) specifically designed to enable users to write scalable multi-threaded programs for CUDA-enabled GPUs [56]. CUDA (the language) was, in essence, an extension of C which provided a fine-grained data parallelism and thread parallelism by offering three key abstractions: (1) a hierarchy of thread groups<sup>9</sup>, (2) shared memories, and (3) barrier<sup>10</sup> synchronization. These were simply exposed to the programmer as a minimal set of language extensions. They allowed the programmer to partition the problem into coarse sub-problems that could be solved independently in parallel by blocks of threads. This decomposition maintained language expressivity by allowing threads to cooperate when solving each sub-problem, and at the same time enabled automatic scalability [55] (see Figure 3.13). Simply put, CUDA brings together several things:

- A massively parallel hardware specifically designed to be able to compile and run generic code, with appropriate drivers<sup>11</sup> for doing so.
- A programming language based on C as the programming tool with only minimal set of language extensions.
- A software development kit that includes libraries, various debugging, profiling and compiling tools, and bindings that let CPU-side programming languages invoke GPU-side code.

The point of CUDA is to write code that can run on compatible massively parallel SIMD architectures. Massively parallel hardware can run a significantly larger number of operations per second than the CPU, at a relatively lower financial cost, yielding performance improvements of 50% or more in situations that allow it.

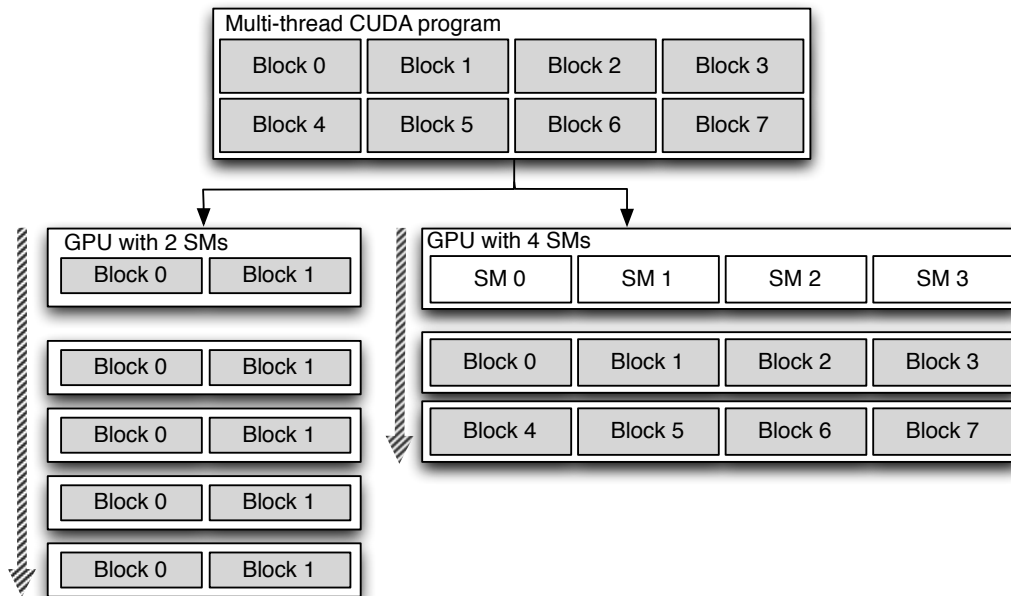
---

<sup>9</sup>In programming languages, thread groups provide a mechanism for collecting multiple threads into a single object and manipulating those threads all at once, rather than individually.

<sup>10</sup>In programming languages, a barrier for a group of threads or processes in the source code means any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier.

<sup>11</sup>A driver is software that allows a computer to communicate with hardware or devices. Without drivers, the hardware connected the computer does not work properly.

**Figure 3.13:** Automatic scalability of CUDA architecture. A streaming multiprocessor (SM) contains multiple cores that can be run simultaneously. A thread block is a set of concurrent threads. Taken from *CUDA programming guide* [55].



One of the benefits of CUDA over the earlier methods is that a general-purpose language is available. So instead of having to use complex techniques to emulate general-purpose computers, the developers can use CUDA programming language, which is based on C with a few additional keywords and concepts, which makes it fairly easy for non-GPU programmers to pick up.

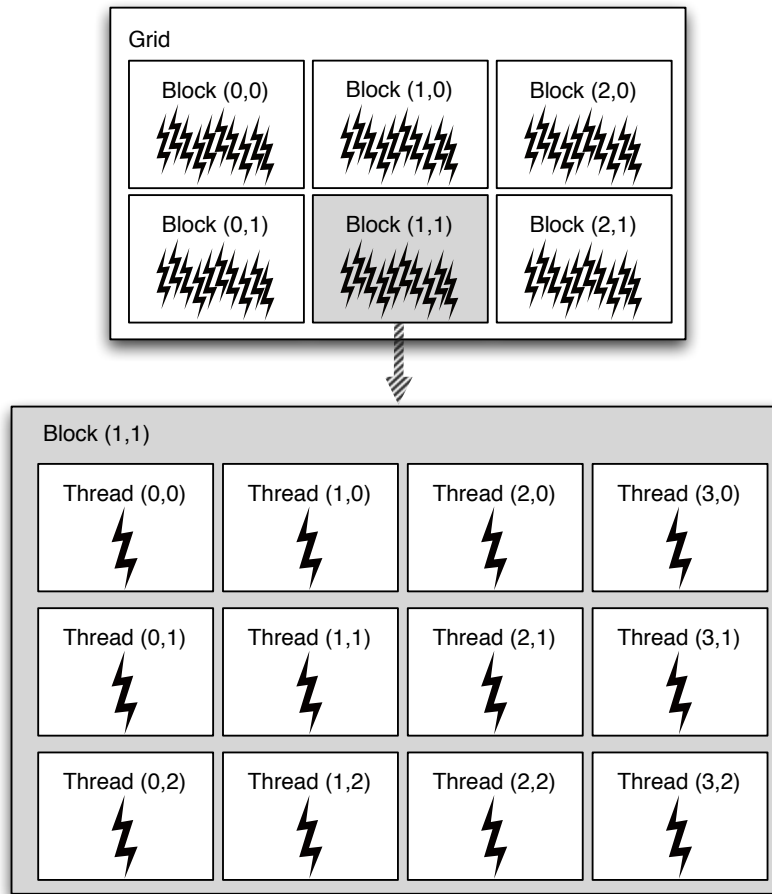
The simplicity of CUDA stems from its architecture. CUDA programs contain a sequential part, called the kernel program. The kernel is written in conventional C-code. It represents the operations to be performed by a single thread on the GPU (hereon for the rest of this document GPU and its memory will be referred to as the device) instead of the CPU (from this point until the end of this document the CPU and its memory will be referred to as the host). The kernel is invoked as a set of concurrently executing threads. These threads are organized in a hierarchy consisting of thread blocks and grids (see Figure 3.14). All threads run the same code and each has an ID that it uses to compute memory addresses and make control decisions. The kernel launches a grid of thread blocks. A thread block is a set of concurrent threads (with associated unique IDs) and a grid is a set of independent thread blocks. Many problems are naturally described in a flat, linear style mimicking the model of C's memory layout. However, other tasks, especially those encountered in the computational sciences, are naturally embedded in two or three dimensions. Hence threads and block IDs are used to simplify memory addressing when processing multidimensional data. See Figure 3.14.

The main reason for this hierarchical organization is the advantages it offers for thread communication and synchronization. Threads within a thread block can communicate through a per-block shared memory (PBSM) and may synchronize using barriers. However, threads located in different blocks cannot commu-



nicate or synchronize directly [57]. As it illustrated in Figure 3.13, the CUDA applications use a scalable processor array. The array consists of a number of streaming multiprocessors (SMs). Each SM contains a number scalar processors (SPs) that share a per block shared memory (PBSM). All threads of a thread block are executed concurrently on a single SM. The SM executes threads in small groups, called warps, in single-instruction multiple-thread (SIMD) fashion. Thus, parallel performance is generally penalized by data-dependent conditional branches and improves if all threads in a warp follow the same execution path.

**Figure 3.14:** Grid of thread blocks. A thread block is a set of concurrent threads (with associated unique IDs) and a grid is a set of independent thread blocks. Taken from *CUDA programming guide* [55].



One of the most important abstractions that CUDA offers is allowing developers to use different types of shared memories. Therefore, in order to write an efficient CUDA application, it is important to understand the different types of memory spaces. Besides the PBSM, there are six other types of memory (see Figure 3.15):

**Readable and writable global memory:** This memory is relatively large, but has high latency, low bandwidth, and is not cached. The effective bandwidth of global memory depends heavily on the

memory access pattern, e.g. coalesced access generally improves bandwidth. This memory can also be accessed by the CPU.

**Readable and writable per-thread local memory:** This memory is of limited size and is not cached. Access to local memory is as expensive as access to global memory and is always coalesced.

**Read-only constant memory:** This memory is of limited size and cached. The reading cost scales with the number of different addresses read by all threads. Reading from constant memory can be as fast as reading from a register (e.g. if all threads of a half-warp read the same address).

**Read-only texture memory:** This memory is large (depending on the size of global memory) and is cached. Texture memory can be read from kernels using texture fetching device functions. Reading from texture memory is generally (not absolutely) faster than reading from global or local memory.

**Readable and writable per-block shared memory:** This memory is fast on-chip memory of limited size. Shared memory can be accessed by all threads in a thread block. Shared memory is divided into equally-sized banks that can be accessed simultaneously by each thread. Accessing the shared memory can be as fast as accessing a register.

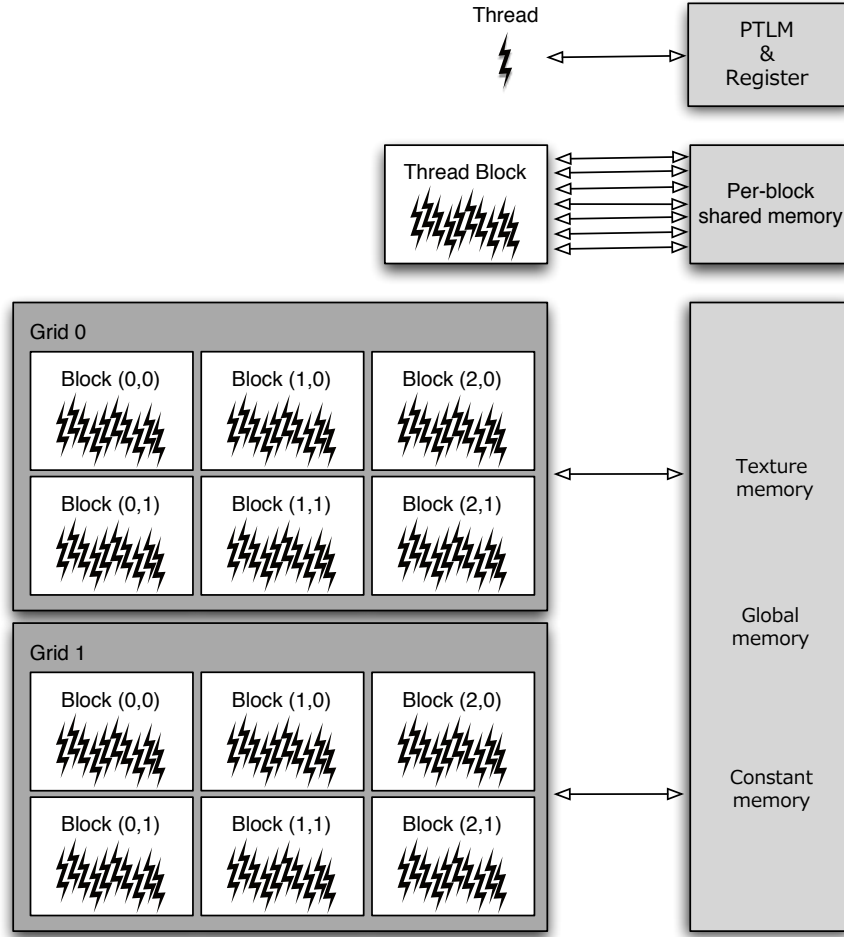
**Readable and writable per-thread registers:** These registers are the fastest memory to access but is of very limited size. The total number of registers and registers per thread depend on the model and the architecture of the hardware in use.

It's noteworthy that the sizes of the above-mentioned memories depend on the type of architecture (e.g. sizes vary between Fermi and Kepler architectures). Further, these memory sizes vary between models within a particular architecture.

### 3.4.3 Kepler vs. Fermi architecture

Kepler is the codename for the latest GPU architecture developed by NVIDIA as the successor to the Fermi architecture. NVIDIA's Kepler architecture is built on the foundation of NVIDIA's Fermi GPU architecture. A major difference in the architectures is in the number of CUDA cores per SM, which has been increased to 192. The Kepler-based GPUs have a total of 8 streaming multiprocessors, making a total of 1536 CUDA cores in contrast to 512 cores of Fermi-based GPUs. Despite tripling the number of cores, the physical die size is about two thirds smaller than Fermi, and has just 500 million more transistors (3.5 billion compared to 3 billion) [58]. Table 3.2 highlights some the more important differences between these two architectures.

**Figure 3.15:** Memory hierarchy of CUDA. PTLM represents per thread local memory. Taken from *CUDA programming guide* [55].



**Table 3.2:** Major differences between Kepler and Fermi architectures. Most popular models of these two architecture (GTX 580 representing Fermi and GTX 680 representing Kepler).

Feature	Fermi (GTX 580)	Kepler (GTX 680)
Power consumption	244 Watts	195 Watts
CUDA cores	512	1536
Compute Capability	2.1	3.1
Streaming Multiprocessors (SMs)	16	8
Cores per SM	32	192
Wrap scheduler per SM	2	4
Cache per SM	64 KB	512 KB
Memory bandwidth	192.2 GB/s	192.2 GB/s
Core speed	772 MHz	1002 MHz

# CHAPTER 4

## COMPARISON OF ASSEMBLY SOFTWARE

### FOR METAGENOMIC DATA

The emergence of next-generation sequencing platforms has led to resurgence of research in whole-genome shotgun assembly algorithms and software. DNA sequencing data from these platforms typically present shorter read lengths, higher coverage, and different error profiles compared with Sanger sequencing data. Over the last years, several assembly software packages have been created or revised specifically for *de novo* assembly of next-generation sequencing data. This chapter provides an evaluation of six popular *de novo* assembly software tools, three of which (IDBA-UD [59], RayMéta [19], and MetaVelvet [60]) were specifically designed for metagenomics data, and three of which (Newbler [13], SSAKE [61], and MIRA [12]) were not. The accuracy, performance, and computational requirements of these assemblers are evaluated using three datasets of simulated sequence reads, each having a different community complexity (low, medium, or high), as well as real reads obtained from the sequencing of environmental samples using Ion Torrent technology.

## 4.1 Methods

### 4.1.1 Artificial metagenomic communities

The artificial communities generated for this study were roughly based on the work of Pignatelli et al. [62]. In their study, communities were composed of organisms whose full genome sequences have been determined. Four synthetic datasets were created representing different community complexities and sequencing depths: low complexity (LC), a set where most reads were drawn from a pool of small number of dominant organisms; medium complexity (MC), having a greater number of dominant organisms; high complexity (HC), having no dominant organisms. In this study, we focused on sequencing data from the Ion Torrent sequencing platform, which typically generates approximately 9 million reads in a single run, making this an appropriate value for the number of reads from each community. Given this, here we use LC, MC, and HC communities, each having approximately eight million corresponding reads.

The study by Pignatelli et al. used only approximately 113 organisms, far fewer than would likely be found in a real metagenomics sample. Here we use all viral ( $n = 1396$ ), algal ( $n = 2$ ), fungal ( $n = 34$ ), and bacterial/archaeal ( $n = 2155$ ) genomes sequenced as of July 2,14 and publicly available at the NCBI

Reference Sequence database (RefSeq) [63], giving a total of 3591 organisms. We sought to create artificial communities with this larger set of organisms but with similar abundance distributions to those used by Pignatelli and co-authors. This was performed as follows. For a given community (say, LC), the organism frequencies as given by Pignatelli were sorted from largest to smallest frequency. A scatterplot was created with the  $x$  axis indicating the rank of the organism’s frequency (with 1 being the most frequent), and the  $y$  axis indicating the frequency of the organism with that rank. This curve was best approximated using a power function ( $y = ax^b$ ). Values of  $a$  and  $b$  were determined that resulted in a curve closely approximating the distribution of that community given by Pignatelli et al. Each organism from our larger set was then arbitrarily assigned a unique rank  $x$  between 1 and 3591, and the relative frequency of that organism was calculated as  $y = ax^b / \sum_{x=1}^{3587} ax^b$ . The denominator was introduced so that the frequencies sum to 1.

#### 4.1.2 Generation of artificial reads

In a real metagenomics study, DNA would be extracted from a sample and sequenced, with the resultant reads subjected to quality control before being used as input to a sequence assembly program. In this study, artificial reads were generated in a manner designed to match as closely as possible to real pre-quality control reads in terms of quantity, length distribution, and quality distribution, with this set of reads being subsequently filtered based on quality and length. To achieve this, DNA was extracted from a sediment sample from an in-progress metagenomics study (manuscripts in preparation), which was then sequenced using the 316 chip on an Ion Torrent Personal Genome Machine (Life Technologies). These reads were used as input to Better Emulation of Artificial Reads (BEAR) [64], a custom program for generating artificial reads. BEAR determines the error rate, error type, length and the quality distributions of the reads it is given as input, and then generates artificial reads drawn from already-sequenced genomes with matching length and quality distributions.

## 4.2 Results

As discussed in Section 3.1, at present, mainly three distinct strategies are applied in short reads assembly. Among them, greedy-extension is the implementation of a string-based method, while de Bruijn graph and overlap-layout-consensus are two different graph-based approaches. We tested three metagenomics-specific assemblers (IDBA-UD, RayMéta, and MetaVelvet), as well as three assemblers designed for the *de novo* assembly of single genomes (Newbler, SSAKE, and MIRA). Details of these assemblers are given in Table 4.1. To simplify comparisons, default parameters were used for all programs. Also, after the assembly process, all contigs that were shorter than the average read length were discarded.

Three simulated datasets of varying species complexity were generated by the sequence simulator program BEAR [64]. Ion Torrent data (377,630 reads in total) were used to train the read length and quality score models. Each dataset consisted of  $\sim 8$  million reads sampled from 3591 genomes (2 algae, 2155 bacteria,

**Table 4.1:** List of evaluated sequence assemblers. Desired read length represents the average read length assembler can process

Metagenomic-specific?	Assembler name	Assembly Strategy	Desired read length	Reference
Yes	IDBA-UD	de Bruijn graph	~(25-125) bp	[59]
	RayMéta	de Bruijn graph	~(25-125) bp	[19]
	MetaVelvet	de Bruijn graph	~(25-125) bp	[60]
No	Newbler	OLC	~(450-1000) bp	[13]
	MIRA	OLC	~(25-1000) bp	[12]
	SSAKE	Greedy Extension	~(25-125) bp	[61]

34 fungi, and 1396 viruses). Three datasets were created representing different community complexities and sequencing depths: low complexity (LC), a set where most reads were drawn from a small number of dominant organisms; medium complexity (MC), having a greater number of dominant organisms; and high complexity (HC), having no dominant organisms. Real data was obtained from sediment samples using whole genome shotgun sequencing via an Ion Torrent Personal Genome Machine. Table 4.2 shows the characteristics of the data provided as input for the assemblers evaluated in this paper. It must be mentioned that the issue of filtering sequence real WGS data, based on the FastQ [65] quality scores provided by the Ion Torrent sequencer was investigated. Because Ion Torrent has been shown to underestimate quality scores [66], several quality thresholds were chosen (e.g., Q17, Q20, and Q25). Because the reported quality of Ion Torrent PGM (Personal Genome Machine) data is typically much lower than 454 data [66], using the standard quality threshold of Q25 leads to most of the data being discarded. As such, the following quality filtering steps were performed in several combinations and Q17 quality threshold provided the best results:

- Remove Torrent adapter and discard any reads shorter than 100 bp after adapter trimming.
- Remove any reads with an average quality less than the given threshold.
- Trim reads to the length where the average quality of all reads falls below the given threshold.

**Table 4.2:** Characteristics of input data

	Simulated Data			Real Data
	Low Complexity	Medium Complexity	High Complexity	Env. Samples
Reads Total Size (Bp)	1,228,297,506	1,228,033,188	1,228,189,696	1,654,227,752
Average read size	159	159	159	193
Size of the longest read	176	176	176	220
Number of the reads	7,724,359	7,721,976	7,722,812	8,567,699

Several metrics were used to evaluate the assemblers with respect to the number of contigs generated and the lengths of those contigs. These were number of contigs, average contig length, aggregate contig size,

number of contigs longer than ten times the average read size, size of the largest contig, N50 (the size  $N$  such that 50% of the bases in the contigs are of size  $N$  or greater), and N80 (analogous to N50).

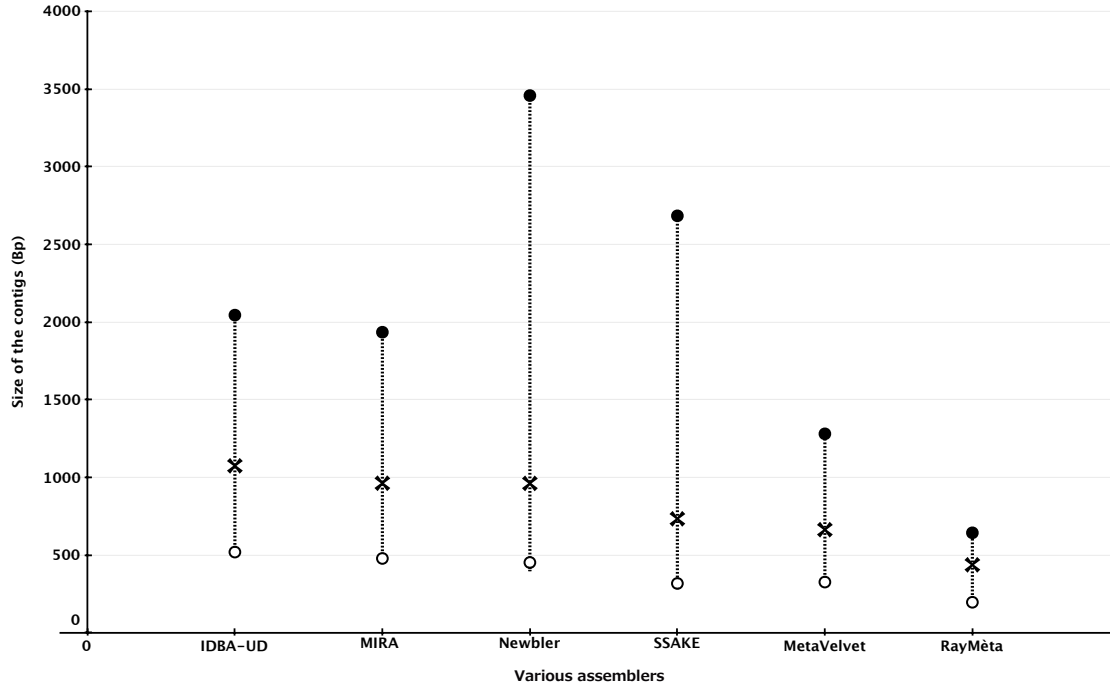
After assembling the three artificial datasets, the accuracy of each of the resultant contigs was evaluated by mapping it back to the reference genomes. This was done using a combination of Bowtie 2 [67] and BLAST [26]. Preliminary testing revealed that Bowtie 2 was extremely fast; however, it occasionally lacked of sensitivity (i.e. it would not report a match for some contigs, especially shorter ones, even though BLAST would report a very close match). As such, Bowtie 2 was first used to search for each contig in the reference genomes. Any contigs having no match were then searched used as BLAST queries. The quality of all matches was based on the “normalized edit distance”,  $(L - E)/L \times 100\%$ , where  $E$  is the edit distance and  $L$  is the length of the contig. For Bowtie 2,  $E$  was taken directly from the Bowtie 2 output. For BLAST,  $E$  was calculated by multiplying the percent identity according to BLAST by the proportion of the full-length contig that was involved in the BLAST local alignment.

Each assembler was evaluated according to three main criteria: computational resources used, the number and lengths of contigs generated, and the accuracy of generated contigs (only for simulated data). All assembly runs were performed on a machine with Ubuntu Linux as the operating system, with two 2.4 GHz 8-core processors and 384 GB of memory. To ensure comparability between assembler outputs, singletons<sup>1</sup> and contigs less than the average read size were discarded before subsequent analyses. This is because not all assemblers keep singletons and/or contigs shorter than 100 bp. Figures 4.1, 4.3, 4.5, and 4.7 illustrate the size statistics of all evaluated algorithms for datasets with varying complexities. Tables 4.3 - 4.6 show the CPU details of different assembly software assembling various datasets. Finally, Figures 4.2, 4.4, and 4.6 present the percentage of contigs produced by various assembly software mapping to the reference genome. The results presented in the aforementioned figures and tables are discussed in Section 4.3. For more detailed information about the results, please refer to Comparison of assembly software (Appendix A).

---

<sup>1</sup>A singleton read is a read that did not show any significant overlap with any other reads

**Figure 4.1:** Size statistics of various assembly programs with low complexity simulated data. “●” represents N50, “×” represents average config size and “○” represents the N80.



**Table 4.3:** CPU details of various assembly software, assembling low complexity simulated data.

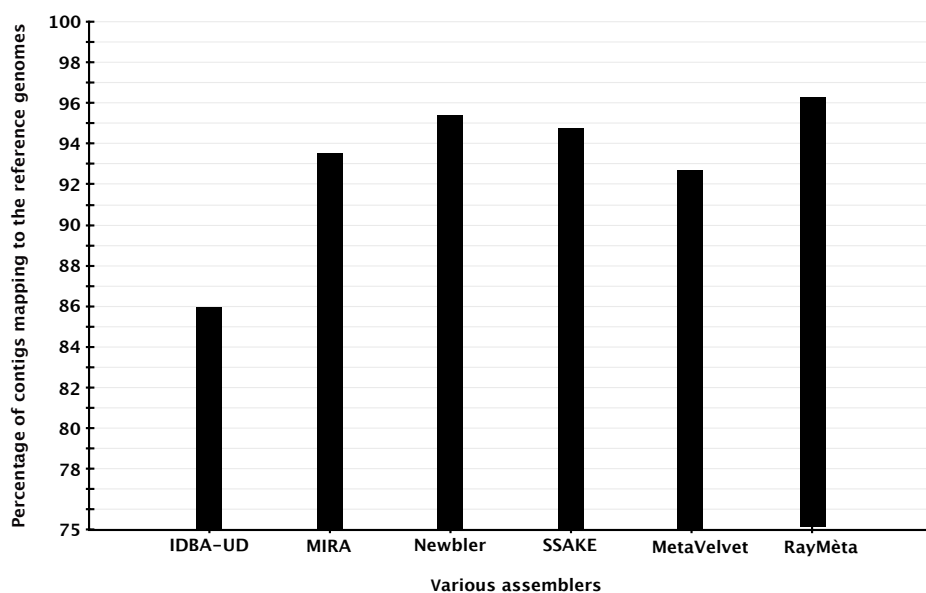
Software	Elapsed real time	User time <sup>a</sup>	System time <sup>b</sup>	Max. RAM Occupancy
IDBA-UD	35 m 30 s	231 m 47 s	1 m 12 s	8,426,048 KB
MIRA	1963 m 51 s	2392 m 0 s	157 m 17 s	45,230,832 KB
Newbler	79 m 40 s	70 m 38 s	1 m 4 s	10,187,564 KB
SSAKE	161 m 37 s	159 m 44 s	0 m 54 s	12,597,256 KB
MetaVelvet	16 m 7 s	15 m 22 s	0 m 13 s	8,232,240 KB
RayMéta	121 m 12 s	723 m 13 s	244 m 38 s	1,322,376 KB

<sup>a</sup>Total amount of CPU-time that the process spent in user mode.

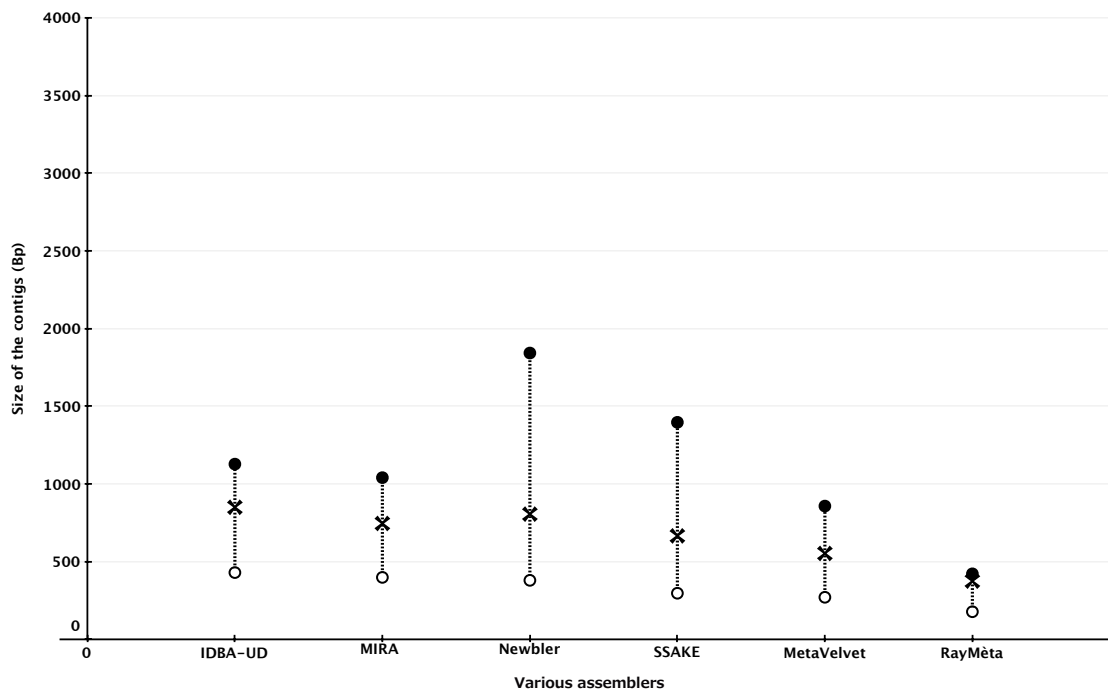
<sup>b</sup>Total amount of CPU-time that the process spent in kernel mode



**Figure 4.2:** Percentage of contigs produced by various assembly software, assembling low complexity simulated data.



**Figure 4.3:** Size statistics of various assembly programs with medium complexity simulated data. “•” represents N50, “x” represents average config size and “o” represents the N80.



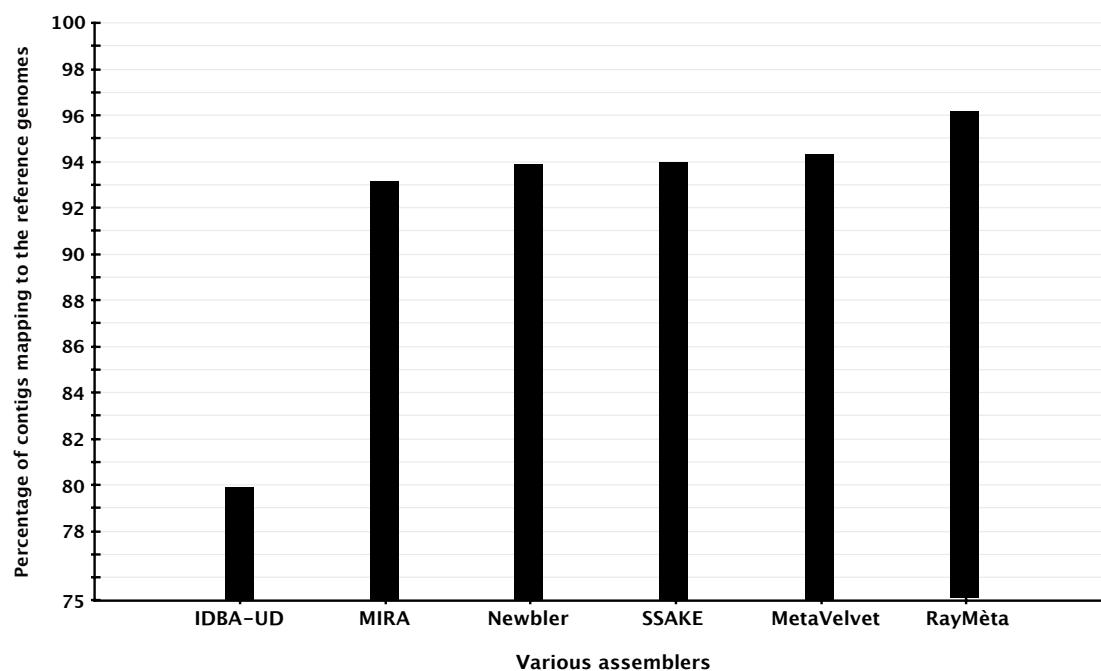
**Table 4.4:** CPU details of various assembly software, assembling medium complexity simulated data.

Software	Elapsed real time	User time <sup>a</sup>	System time <sup>b</sup>	Max. RAM Occupancy
IDBA-UD	42 m 0 s	271 m 24 s	1 m 47 s	11,260,668 KB
MIRA	954 m 36 s	1251 m 2 s	60 m 40 s	45,226,744 KB
Newbler	104 m 36 s	92 m 38 s	1 m 51 s	10,187,564 KB
SSAKE	265 m 34 s	263 m 38 s	0 m 46 s	13,160,892 KB
MetaVelvet	17 m 38 s	16 m 43 s	0 m 16 s	10,380,080 KB
RayMéta	153 m 33 s	897 m 52 s	328 m 27 s	3,008,824 KB

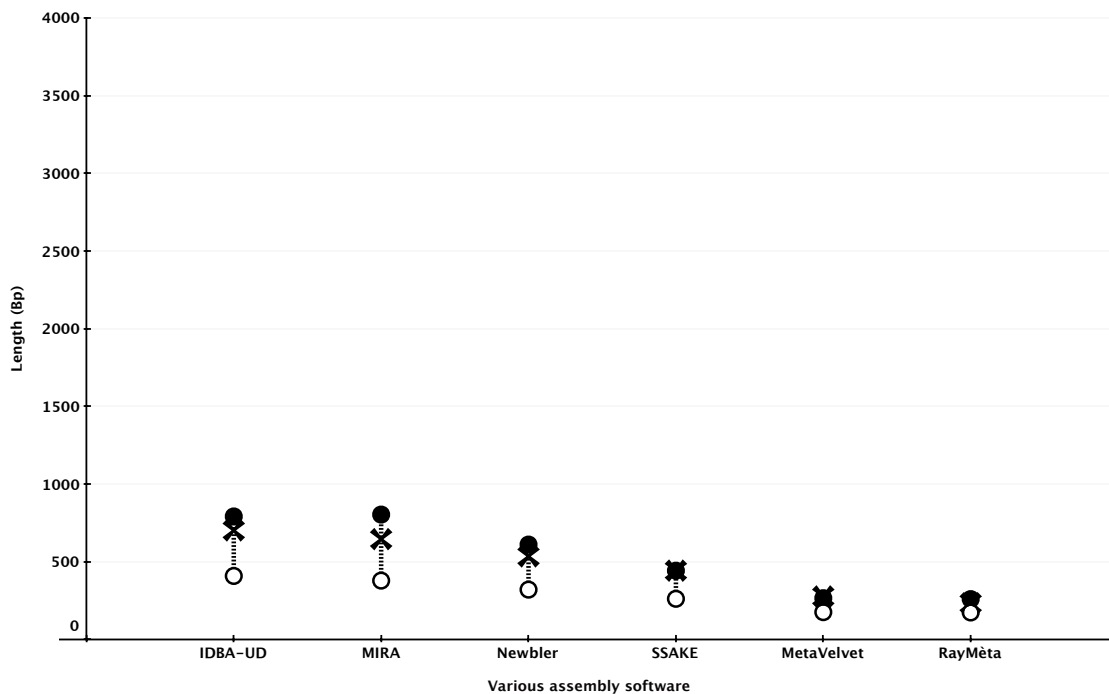
<sup>a</sup>Total amount of CPU-time that the process spent in user mode.

<sup>b</sup>Total amount of CPU-time that the process spent in kernel mode

**Figure 4.4:** Percentage of contigs produced by various assembly software, assembling medium complexity simulated data.



**Figure 4.5:** Size statistics of various assembly programs with high complexity simulated data. “●” represents N50, “×” represents average config size and “○” represents the N80.



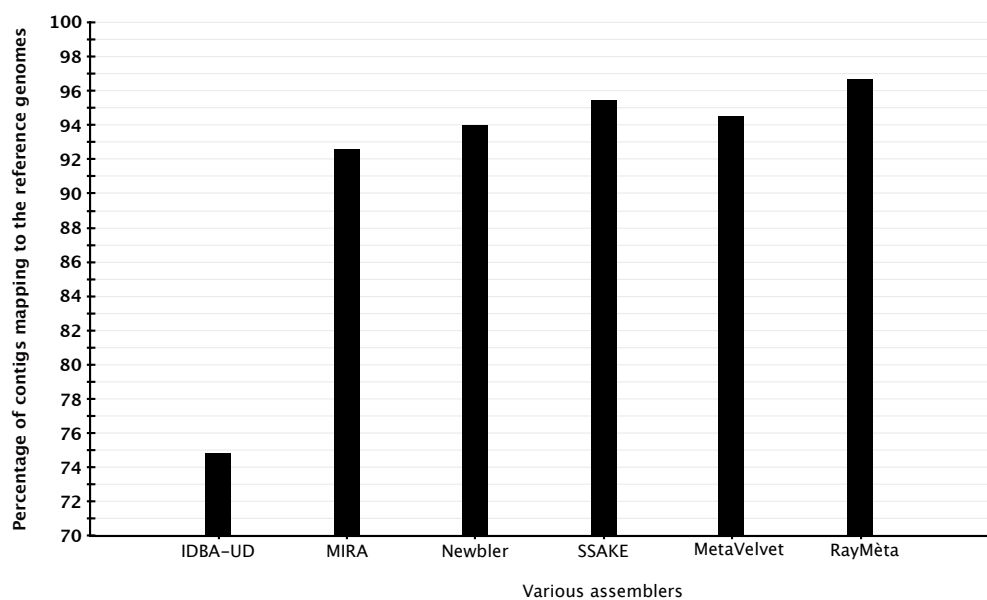
**Table 4.5:** CPU details of various assembly software, assembling high complexity simulated data.

Software	Elapsed real time	User time <sup>a</sup>	System time <sup>b</sup>	Max. RAM Occupancy
IDBA-UD	64 m 31 s	413 m 17 s	3 m 27 s	12,527,536 KB
MIRA	700 m 22 s	784 m 2 s	25 m 43 s	42,873,644 KB
Newbler	95 m 34 s	84 m 5 s	1 m 52 s	10,187,564 KB
SSAKE	536 m 59 s	534 m 12 s	0 m 58 s	14,817,732 KB
MetaVelvet	24 m 59 s	23 m 50 s	0 m 25 s	20,754,788 KB
RayMéta	292 m 53 s	1760 m 29 s	579 m 11 s	45,227,760 KB

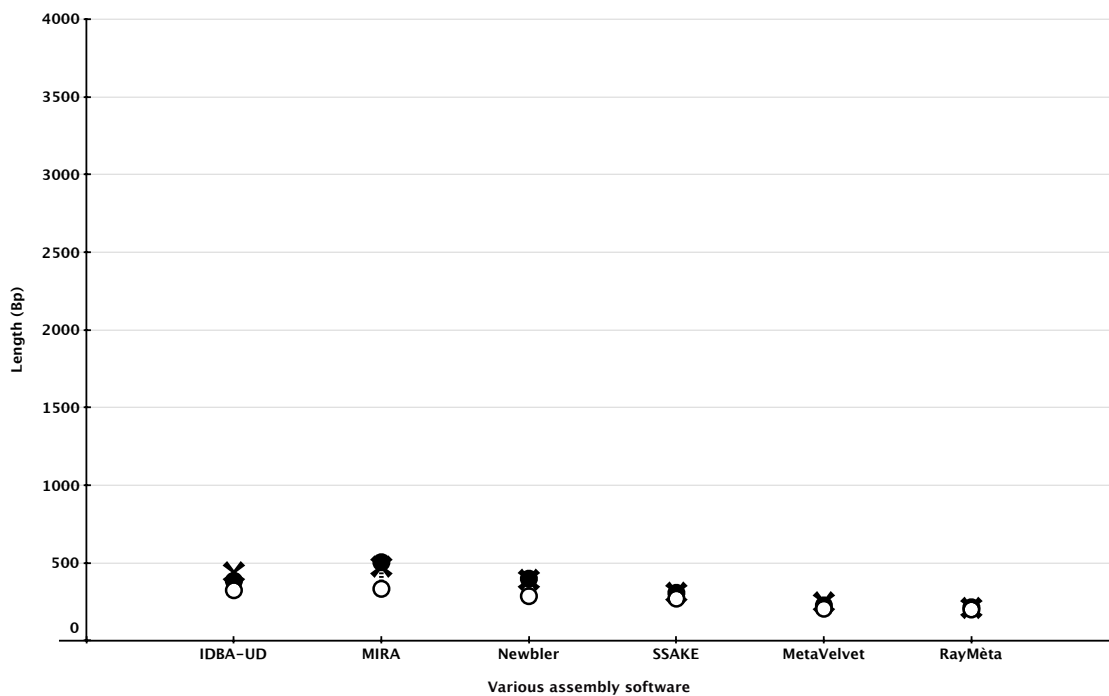
<sup>a</sup>Total amount of CPU-time that the process spent in user mode.

<sup>b</sup>Total amount of CPU-time that the process spent in kernel mode

**Figure 4.6:** Percentage of contigs produced by various assembly software, assembling high complexity simulated data.



**Figure 4.7:** Size statistics of various assembly programs with real data. “●” represents N50, “×” represents average config size and “o” represents the N80.



**Table 4.6:** CPU details of various assembly software, assembling real WGS data.

Software	Elapsed real time	User time <sup>a</sup>	System time <sup>b</sup>	Max. RAM Occupancy
IDBA-UD	110 m 22s	682 m 34 s	4 m 54 s	31,676,264 KB
MIRA	427 m 46 s	448 m 50 s	15 m 16 s	57,129,692 KB
Newbler	141 m 58 s	137 m 43 s	2 m 9 s	7,808,088 KB
SSAKE	1092 m 16 s	1088 m 6 s	1 m 10 s	17,762,700 KB
MetaVelvet	59 m 42 s	58 m 26 s	0 m 41 s	48,910,948 KB
RayMéta	907 m 48 s	568 m 49 s	1573 m 40 s	14,806,740 KB

<sup>a</sup>Total amount of CPU-time that the process spent in user mode.

<sup>b</sup>Total amount of CPU-time that the process spent in kernel mode

### 4.3 Conclusions

Although no single assembler performed best on all our criteria, Newbler gave the longest contigs and the largest N50 and N80 values as well as a very good correctness score (percentage of contigs mapping to the reference genomes). Meanwhile Newbler’s aggregated assembly sizes were small. On the other hand, although slow, MIRA scored very well in size statistics and correctness analysis, and produced very large aggregate contig sizes. Although the IDBA-UD assemblages had large sizes, they had the worst correctness scores. The remaining assemblers all performed almost equally well, with the exception of RayMéta, which had the worst size statistics. The performance of the assemblers was especially poor when assembling environmental samples. All in all, our evaluation of assemblers suggested that although no single assembler performed best on all of our criteria, MIRA slightly outperformed the other programs.

# CHAPTER 5

## DATA AND METHODOLOGY

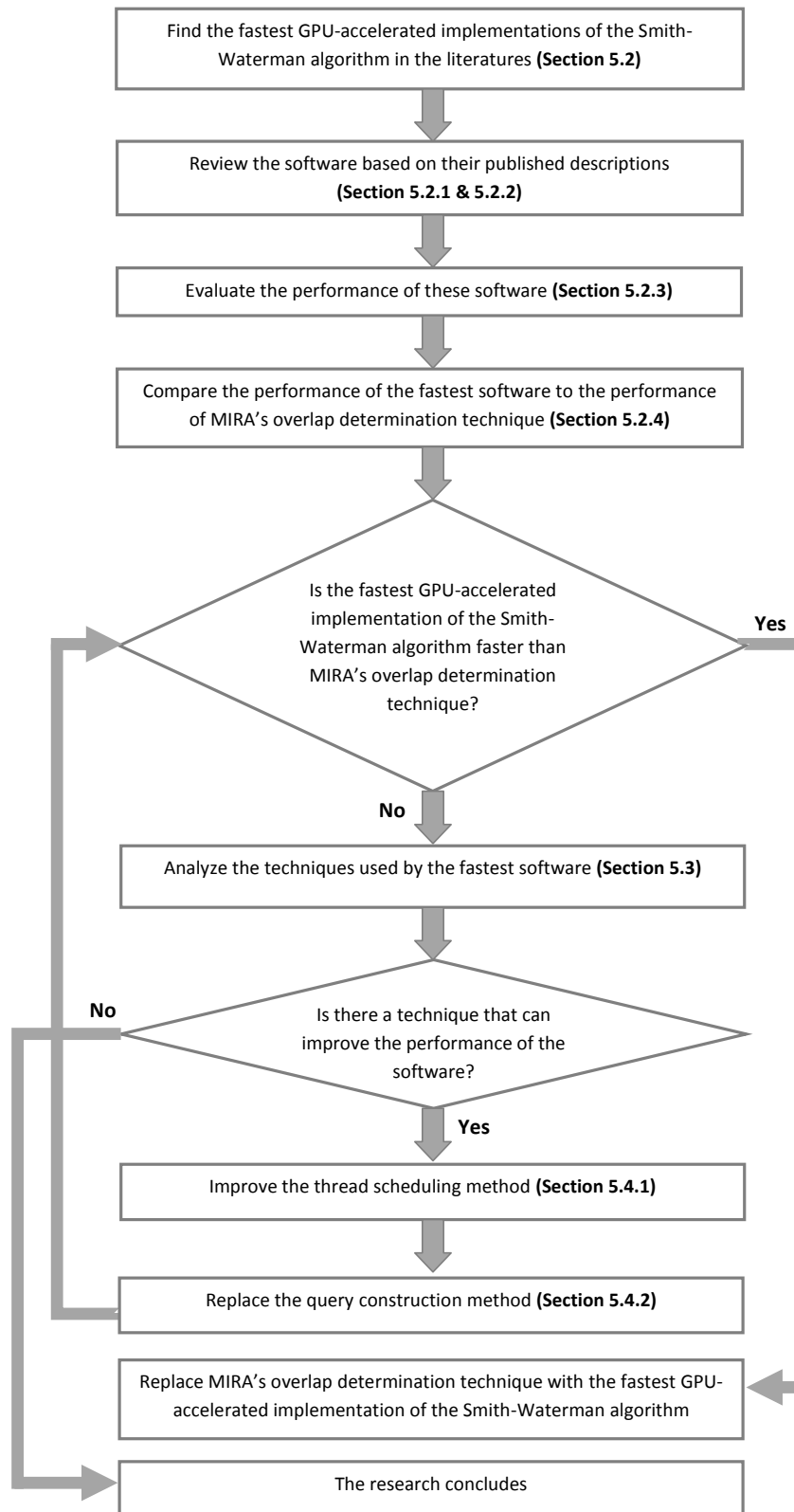
This section describes the methodology used to perform the two main analyses completed in this thesis: selection of an existing GPU-accelerated implementation of the Smith-Waterman algorithm that meets a number of criteria such that modification and improvement of the program is possible (Section 5.2 and Section 5.3), and optimization of the selected program and its adaptation for aligning medium-length metagenomic reads (Section 5.4). Since some of the steps in this chapter are dependent on results generated in the previous chapters, a brief background on the subject is provided before each section to maximize readers' comprehension of the methodology used. Some results fully reported in Chapter 6 are also previewed in this chapter as the discussion is dependent on them.

### 5.1 Structure of this chapter

This chapter briefly reviews the most important principles of sequence alignment, common dynamic programming techniques that are used in this domain, the alternatives for these algorithms and their relative advantages and disadvantages. Based on the comparison results presented in Chapter 4 and the background provided in Chapter 3, a detailed discussion is provided about the efforts made to reduce the runtime of sequence alignment algorithms running on commodity PC hardware (Sections 5.2.1 and 5.2.2). This discussion will set the stage for identifying a set of GPU-accelerated alignment software that can be used to accurately find potential overlaps between each pair of sequences from a given list in a timely manner. The suitability of these software tools is determined based on the review of their published description and the evaluation of their performance presented in Section 5.2.3.

After identifying the fastest sequence aligner, the performance of this program is benchmarked against the performance of MIRA's overlap determination stage (Section 5.2.4). The results of this comparison prompts a close analysis of the chosen sequence aligner in order to identify its technical features. Section 5.3 describes these techniques. Some of these techniques will be further extended to improve the performance of this software tool and some will be replaced. Full discussion on this topic can be found in Section 5.4. The methodology followed in this research is presented in Figure 5.1.

**Figure 5.1:** Methodology flow chart



## 5.2 Selection of the fittest

### 5.2.1 Related works

As presented in Chapter 4, a group of assemblers that are currently used for assembly of medium-length nucleotide data were evaluated. The applicability and performance of these assemblers were compared utilizing simulated data of different microbial community complexities (low, medium and high complexities, abbreviated as LC, MC and HC, respectively) as well as real DNA sequence fragments obtained from randomly selected whole-community DNA using Ion Torrent technology. Considering the computational time, maximum random access memory (RAM) occupancy, assembly accuracy and integrity, the presence of a program's source and its maintainability and modularity, our study identified MIRA as the best potential assembly software that could meet our performance expectations while having the potential for improvement and modifications. For this purpose, several independent sequential modules implemented in MIRA (Figure 1.1) were considered for change with scalable replacements. Eventually, the most compute-intensive portion of the application (i.e. the alignment step using the Smith-Waterman algorithm) was selected for replacement by an alternative approach.

Aknowledging the above consideration, it's fitting to briefly review the Smith-Waterman algorithm and some of the efforts made to increase the performance of this algorithm by exploiting hardware acceleration techniques.

As discussed in Chapter 3, the Smith-Waterman algorithm is commonly used for computing optimal pairwise sequence alignments. Algorithm 1 presents the first portion of a basic Smith-Waterman algorithm, which determines the score of the best alignment, expressed in pseudocode. As illustrated, two sequences, *seqA* and *seqB*, of lengths respectively  $m$  and  $n$ , are compared using the scoring matrix  $W$  and a gap parameter,  $g$ . It is apparent that this algorithm calculates the values in a matrix of size  $m + 1 \times n + 1$ , where one letter of the sequence *seqB* (outer loop) is compared against all letters of the sequence *seqA* (inner loop). For each comparison, three intermediary results are produced in each iteration. The final value of score is computed as the maximum of all the values found in each iteration. To obtain the alignment with that score, one then performs a traceback step. As discussed in Chapter 3, overall complexity of the SW algorithm is  $O(m \times n)$ .

It is noteworthy that the Smith-Waterman algorithm typically implemented using an affine gap penalty scheme, rather than a simple gap penalty scheme illustrated in Algorithm 1. This implementation of the Smith-Waterman algorithm requires the maintenance of 3 matrices instead of 1 (i.e.  $H$ ), but the main idea of the algorithm stays the same [33].

The complexity of the Smith-Waterman algorithm is quadratic with respect to the lengths of alignment targets [18], which makes it time consuming for applications involving large datasets. Therefore, heuristic methods have been introduced to accelerate sequence alignment. The drawback is that the more efficient the



**Input** : Sequences  $seqA$  of length  $m$  and  $n$  of length  $len_b$

**Assumptions:**  $g$  is a gap-scoring scheme and  $W(seqA[i], seqB[j])$  is a similarity function (substitution matrix).

*Initialize matrix  $S$  properly.*

*Build the matrix  $S$  row by row.*

```
for ( $i = 1$  to  $i = m + 1$ ) do
    for ( $j = 1$  to  $j = n + 1$ ) do
         $H[i, j] = \max($ 
             $H[i - 1, j - 1] + W(seqA[i], seqB[j]),$ 
             $H[i - 1, j] + g,$ 
             $H[i, j - 1] + g,$ 
             $0)$ 
        end
    end
end
Return  $H[m, n]$ 
```

**Algorithm 1:** Pseudocode of the first portion of the basic Smith-Waterman algorithm that calculates the score of the best alignment (Focus of this study). The alignment with that score is determined by a subsequent traceback stage.

heuristics, the worse the quality of the result. Another approach to get high-quality results in a short time is to use high-performance computing. Chapter 3.3 briefly discussed how the emergence of accelerator technologies and many-core architectures, such as GPUs, has provided the opportunity to significantly reduce the runtime for many bioinformatics programs, including the Smith-Waterman algorithm, on commonly available and inexpensive hardware.

For instance, Oliver et al. [49, 48] constructed a linear systolic array<sup>1</sup> to perform the Smith-Waterman algorithm on a standard Virtex II FPGA board, achieving a peak performance of about 5 GCUPS (Giga Cell Updates Per Second) using affine gap penalties. Li et al. [68] exploited custom instructions to accelerate the SW algorithm on an Altera Stratix EP1S40 FPGA by dividing the SW matrix into grids of  $8 \times 8$  cells and achieved an estimated peak performance of about 23.8 GCUPS for DNA sequences. Farrar [50] exploited the special instruction set introduced by the Intel Pentium 4 to compute the SW algorithm in a striped pattern, outperforming the previous SIMD based SW implementations by Wozniak [52] and Rognes et al. [53]. This striped SW approach was then optimized for Cell/BE [69]. SWPS3 [54] extends Farrar's work for Cell/BE and also improves it to support multi-core processors. CBESW [70] was designed for the Cell/BE-based PlayStation 3 (PS3) and achieves a peak performance of about 3.6 GCUPS.

---

<sup>1</sup>Systolic arrays are arrays of processors which are connected to a small number of nearest neighbours in a network.

### 5.2.2 GPU-accelerated sequence aligners

The rapidly increasing power of graphics processing units provides the opportunity to significantly reduce runtime for the sequence alignment operation using commodity PC hardware. Modern implementations of the SW algorithm are mainly focused on this new technology. In this section, we briefly review some of the more popular GPU-accelerated implementations of the Smith-Waterman algorithm based on their published descriptions to determine their suitability for the purposes of this study.

The first implementation of the SW algorithm on GPUs was developed by Liu et al. [18] using OpenGL. This work introduced two separate streaming algorithms for dynamic programming (DP)-based biological sequence alignment. Both algorithms took advantage of the particular data dependency relationship in the DP matrix (i.e. matrix  $H$  in Algorithm 1). They were implemented using C++ and OpenGL Shading Language. The implementations of these algorithms achieved speedups of more than an order of magnitude on cheap, readily available graphics hardware.

Although these results were encouraging, the introduction of CUDA and CUDA-enabled GPUs resulted in a simpler and more efficient methodology for performing scientific computing on GPUs. The first implementation of the SW algorithm on NVIDIA CUDA-enabled GPUs was developed by Manavski et al. [51]. By taking advantage of the CUDA architecture which allows direct access to the hardware primitives of the GPU, this method achieved a speedup of more than 3.5 GCUPS running on a workstation with two GeForce GTX 280 cards. The implementation of this method performed 16 times faster than the previous method proposed by Liu et al. [18].

Liu et. al [57] further explored the compute power of CUDA-enabled GPUs. CUDASW++, capable of running on a single-GPU graphics card as well as multi-GPU graphics card, was proposed. An evaluation of the CUDASW++ implementation on a single-GPU NVIDIA GeForce GTX 280 graphics card and a dual-GPU GeForce GTX 295 graphics card illustrated the significant performance improvement of this algorithm compared to other publicly available implementations, such as SWPS3 and CBESW. CUDASW++ supported query sequences of length up to 59K. For protein query sequences ranging in length from 144 to 5,478, the single-GPU version achieved performance between 9.039 and 9.660 GCUPS (average 9.509), and the dual-GPU version achieved performance between 10.660 and 16.087 GCUPS (average 14.484 GCUPS). CUDASW++ 2.0 further optimized the performance of its predecessor. It achieved performance improvement over CUDASW++ 1.0 by as much as 1.77 times with a performance of up to 17 GCUPS on a single-GPU GeForce GTX 280 and 30 GCUPS on an NVIDIA GeForce GTX 295 graphics card [75].

CUDASW++ 3.0 was introduced in 2013 [71] in order to maximize the performance of its predecessors by fully taking advantage of NVIDIA's latest CUDA architecture (i.e. Kepler). CUDASW++ 3.0, written in CUDA C++ and PTX assembly languages, employs the new Streaming Multiprocessor Architecture (SMX) exclusive to GPUs designed based on the Kepler architecture. This software tool attains significant speedups over its predecessor, CUDASW++ 2.0, by taking advantage of CPU and GPU SIMD instructions as well as the concurrent execution on CPUs and GPUs. Evaluation on the Swiss-Prot database shows that

CUDASW++ 3.0 has a performance improvement over CUDASW++ 2.0 up to 2.9 and 3.2 times, with a maximum performance of 119.0 and 185.6 GCUPS for single- and dual-GPU versions, respectively [71].

All of the aforementioned software tools were specifically designed to perform protein database search where one amino acid sequence is aligned with all the other sequences residing in a protein database. None of these software has ever been utilized in a *de novo* assembler nor can any of these tools be readily incorporated into an existing assembler. G-PAS was presented by Frohmberg et al. [72] as a software tool well-tailored for the outlined problem. It therefore can be a viable candidate to be used in the DNA assembly problem. According to the authors, G-PAS performs best for Roche/454 data (177 GCUPS), and even for relatively short sequences from Illumina the algorithm achieves very good performance (112 GCUPS). Tests were performed on a NVIDIA GeForce GTX 580 with 1.5GB of RAM.

The performance results for each software tool presented above are difficult to compare as they depend heavily on the length of sequences, the length deviation of datasets, and the hardware used. Hence, further comparison is needed in order to choose the best GPU-based implementation of the Smith-Waterman algorithm for this work.

### 5.2.3 Comparing GPU-accelerated sequence alignment tools

As evident in the above review, search speed is often reported in GCUPS, which indicates the billions (giga) of cells in the alignment matrix (see Section 3.4.2) processed per second. This metric was hence used to measure the performance of the following algorithms: CUDASW++ 3.0 (v3.0.14) CUDASW++ 2.0 (v2.0.10), G-PAS(v2.0) and SW-CUDA (v1.92).

Considering that all the participating software tools were originally designed to accelerate protein database searches, a different scoring scheme was required for working on nucleotide sequences. After modifying the above mentioned software tools, the performance of these algorithms was benchmarked using simulated datasets as well as real WGS data. Figure 5.1 depicts this step as one of the steps in the methodology followed in this thesis.

To this end, three simulated datasets of varying sizes were generated by the sequence simulator program BEAR [64]. Ion Torrent data (377,630 reads in total) were used to train the read length and quality score models of this simulator. Each dataset consisted of reads sampled from 3591 genomes (2 algae, 2155 bacteria, 34 fungi, and 1396 viruses). Real data was obtained from sediment samples using whole genome shotgun sequencing via an Ion Torrent Personal Genome Machine. These data (i.e. real data) were filtered using the FASTQ quality trimmer program [65]. For more information about the quality filtering of the real WGS data and the process of generating the simulated data, please refer to Section 4.2. Table 5.1 shows the characteristics of the data provided as input to the GPU-accelerated sequence aligners evaluated in this section.

CUDASW++ 2.0 (v2.0.10), G-PAS(v2.0), and SW-CUDA (v1.92) are specifically designed to take advantage of the many-core computing power of NVIDIA's Fermi architecture. The comparison between these

**Table 5.1:** Characteristics of input data

	Simulated Data			Real Data		
Dataset Name	SYN_1000.fna	SYN_10K.fna	SYN_100K.fna	ENV_1000.fna	ENV_10K.fna	ENV_100K.fna
Reads Total Size (Bp)	197,919	1,978,585	19,710,973	181,980	1,897,797	19,388,890
Average read size	197	197	197	181	189	193
Size of the longest read	258	258	258	220	220	220
Number of the reads	1000	10,000	100,000	1000	10,000	100,000
Standard deviation of read lengths	46	45	45	43	38	36

software tools and EMBOSS water was hence needed to be carried out on a machine with a Fermi architecture-based GPUs card. A Lenovo M91p Tower with an Intel Core i7-2600 Quad Core (3.40/3.80GHz, 8MB Intel Smart Cache, 1333MHz FSB) CPU, 4GB RAM and a NVIDIA GeForce GT 640 GPU (797MHz engine clock, 891 MHz memory clock, 2GB DDR3 memory with 128-bit bandwidth) with 384 CUDA cores was then chosen for this purpose. Section 6.1.2 describes the results of the evaluation of the performance of these algorithms.

The fastest algorithm among CUDASW++ 2.0, G-PAS and SW-CUDA was then modified to run on a machine equipped with a GPU with Kepler architecture. The performance of this algorithm was compared to CUDASW++ 3.0 and EMBOSS water on a single NVIDIA GeForce GTX 680 graphics card (1006MHz engine clock, 1502 MHz memory clock, and 2GB DDR5 RAM with 256-bit bandwidth) with 30 SMs (Streaming Multiprocessors) comprising 1536 CUDA cores sharing a configurable 64 KB on-chip memory installed in a Mac Pro with a 2.66 Quad core Intel Xeon CPU (L2 Cache (per Core): 256 KB, and L3 Cache: 8 MB) and 16 GB RAM. Section 6.1.2 reports the results of this comparison.

The EMBOSS sequential implementation of the Smith-Waterman (i.e. water [73]) algorithm is considered to be the *de facto* standard implementation of this algorithm by many researchers. The performance of this program aligning datasets of various sizes is reported in all the evaluations as the base line. The resultant alignment scores of all the benchmarked algorithms were also compared against the alignment scores produced by water and all reported the same score.

As it is fully presented in Section 6.1.2, CUDASW++ 3.0 considerably outperforms the other algorithms. It is worth mentioning that the reported execution times used to calculate the GCUPS include the transfer time of the query sequences from host to GPU, the calculation time of the SW algorithm, and the time taken to transfer back the scores.

#### 5.2.4 Is CUDASW++ 3.0 fast enough?

To answer the question of whether or not it is possible to improve the performance of an OLC-based assembler by using a GPU-based acceleration technique, the performance of MIRA's overlap determination stage was compared against the performance of the fastest GPU-based implementation of the Smith-Waterman

algorithm. Tables 6.1 and 6.2 illustrate the results of this comparison.

It is important to note that due to lack of documentation and inadequate modularity in the software design, MIRA’s overlap determination stage could not be extracted from MIRA package. The results of the comparison clearly show the inadequate performance of CUDASW++ 3.0 in comparison to MIRA. The performance of CUDASW++ 3.0 hence needed to be improved. Figure 5.1 illustrates the process of this improvement step by step.

## 5.3 Improving the fittest

Considering the results of the evaluations outlined in Sections 5.2.3 and 5.2.4, this section describes in detail the structure of CUDASW++ 3.0, exploring the technical features of this algorithm that make it far faster compared to the other benchmarked software. Some of these techniques are further enhanced or completely replaced to improve the performance of this software tool. Considering the specific characteristics of this study’s target data (i.e medium-length (meta)genomic data), a goal of this thesis work is set to improve the performance of CUDASW++ 3.0, while maintaining its accuracy, when aligning medium-length sequences. The correctness of the resultant software is confirmed by comparing against the EMBOSS water program.

### 5.3.1 CUDASW++ 3.0

In the abstract, CUDASW++ 3.0 gains higher speed by benefiting from the faster, more efficient Kepler architecture, its optimized GPU SIMD instructions and the concurrent CPU and GPU computations. This algorithm works in four stages (see Figure 5.2):

1. Distribution of workloads over CPUs and GPUs according to their computing power.
2. Concurrent CPU and GPU computations.
3. Re-computation of all alignments that have exceeded the accuracy threshold using CPU SIMD instructions.
4. Sorting of all alignment scores in descending order and the output of results.

#### Workload distribution

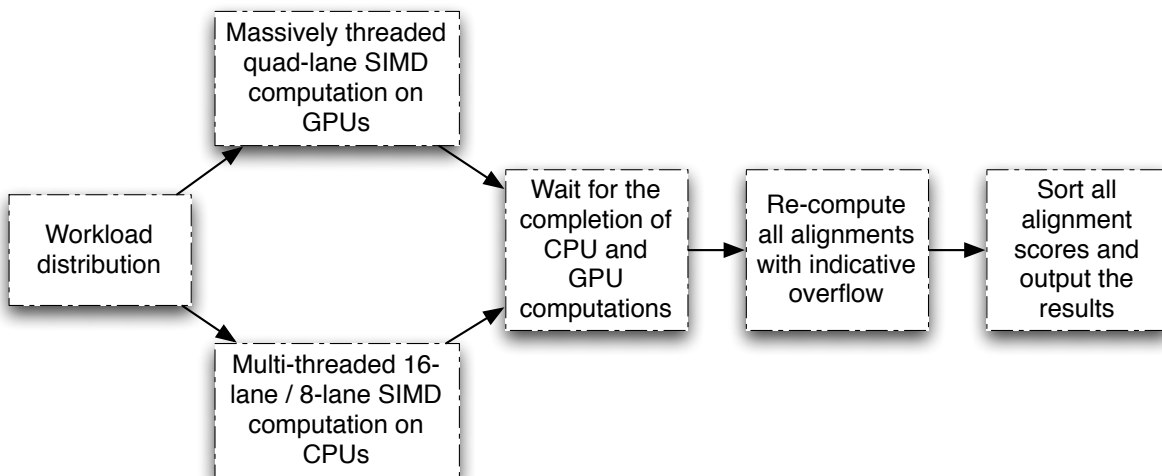
To balance the runtimes between the CPU and GPU SIMD computation, CUDASW++ 3.0 distributes the workload according to the compute power of CPUs and GPUs as follows:

$$R = \frac{N_G f_G}{N_G f_G + N_C f_C / C} \quad (5.1)$$

where  $f_C$  and  $f_G$  are the core frequencies of CPUs and GPUs,  $N_C$  and  $N_G$  are the number of CPU cores and the number of GPU SMs, and  $C$  is a constant derived from empirical evaluations. This value of  $C$  is set

to set to 3.2 and hardcoded in the original program. The portion equal to  $R$  times of the total number of residues in the dataset is assigned to GPUs. All other sequences are distributed to CPUs.

**Figure 5.2:** Program workflow of CUDASW++ 3.0. This figure is taken from Liu et al. paper [71].



### CPU computation

Following the method introduced in the SWIPE program [74], in the second step of the algorithm the CPU computes the Smith-Waterman algorithm by splitting an SSE vector<sup>2</sup> to 16 lanes with 8-bit lane width using only a 7-bit score range (the 7-bit score range from -128 to -1 (signed numbers) or 128 to 255 (unsigned numbers) is used). An 8-bit range, which would allow the same number (16) of parallel computations as a 7-bit range, and at the same time allow a wider score range, is not used because it is slower. For more information please refer to the paper by Rognes [74]. This allows residues from 16 different database sequences to be processed in parallel. These 16 residues are all simultaneously compared to the same query residue. The operations are carried out using vectors consisting of 16 independent bytes. The 16 residues are fed into sixteen independent channels. When the first of these sixteen database sequences ends, the first residue of the next database sequence is loaded into the channel [74].

All alignments, whose scores have overflow potential, are re-computed using 8-lane SSE vectors with 16-bit lane width. An alignment overflow potential is determined by comparing the alignment score with a score limit calculated by subtracting from 128 the maximum substitution score in the scoring matrix. If the score is greater than a score limit, the alignment is deemed to have an overflow potential and thus requires re-computation. More details about the specific implementation of the SSE-based SW algorithm can be obtained from the paper by Rognes [74].

<sup>2</sup>Streaming SIMD Extensions (SSE) is an SIMD instruction set.

SWIPE uses multiple threads that work on different parts of the sequence database. The number of threads is specified when starting the program and should in general be equal to the number of cores of the computer. For the latest generations of Intel processors with hyper-threading, a number of threads equal to the number of logical cores is usually most effective. Since the workload (i.e. subject sequences assigned to CPUs) is known beforehand, CUDASW++ 3.0 calculates the total number of residues in all assigned subject sequences and equally distributes all residues over all threads using a sequence as a unit. This distribution aims to make each thread hold the same number of residues, but not necessarily receiving the same number of subject sequences [71].

## GPU computation

Arguably the most important source of speedup in CUDASW++ 3.0 is the program’s ability to take advantage of the new NVIDIA Kepler architecture. As discussed in Section 3.4.3, the Kepler architecture offers a new streaming multiprocessor architecture (SMX) which leads to increases in the GPU power efficiency (e.g. two Kepler CUDA cores consume 90% power of one Fermi CUDA core). Consequently the SMX can afford additional processing units to execute a whole warp-per-cycle. As a result, it doubles the CUDA cores from 16 to 32 per CUDA array which in turn leads to significant speed up.

Another source of the performance speedup with the new architecture is the expansion of the per-thread local memory size. During the execution of the Smith-Waterman algorithm intermediate alignment data needs to be recorded. Previous versions of CUDASW++ (i.e version 1.0 and 2.0) used the global memory to support long sequences and store the intermediate results. However, since the Kepler architecture has 512 KB per-thread local memory (as opposed to Fermi architecture’s 56 KB per-thread local memory), sequences as long as 65,536 bp can be supported on GPUs. Hence use of global memory is not required anymore for storing intermediate alignments. This expansion in the size of local memory also resolves the need for specific memory access patterns and coalescing techniques (e.g. cell block division technique<sup>3</sup>) implemented in previous versions of CUDASW++ (i.e versions 1.0 and 2.0).

The basis for the computations of the values in each cell in the alignment matrices are the recurrence relations described in Section 3.2 and illustrated in Appendix B. Rognes [74] showed that this recurrence can be written in as little as ten assembly instructions. This new technique of implementation reduced the need for sophisticated flow control and hid the memory access latency which in turn led to more speedup. NVIDIA Kepler architecture supports inline PTX assembly instructions as a low-level parallel thread execution virtual machine and instruction set architecture. To gain more speedup, CUDASW++ 3.0 employs this new feature and implements the recurrence the recurrence shown in Figure B.1. Appendix B provides more information on this subject.

---

<sup>3</sup>To maximize performance and to reduce the bandwidth demand of global memory, CUDASW++ 1.0 and CUDASW++ 2.0 used the cell block division method for the inter-task parallelization, where the alignment matrix is divided into cell blocks of equal size.

## Virtualized SIMD abstraction

The Smith-Waterman algorithm can be parallelized in a variety of methods and scales using GPUs. Different computation approaches have been introduced in order to parallelize the implementation of Smith-Waterman algorithm. However, all these methods are based on the virtualized single instruction, multiple data (SIMD) abstraction.

A brief review of GPU architecture can assist readers to better understand how virtualizing a warp as an SIMD vector could lead to more speedup.

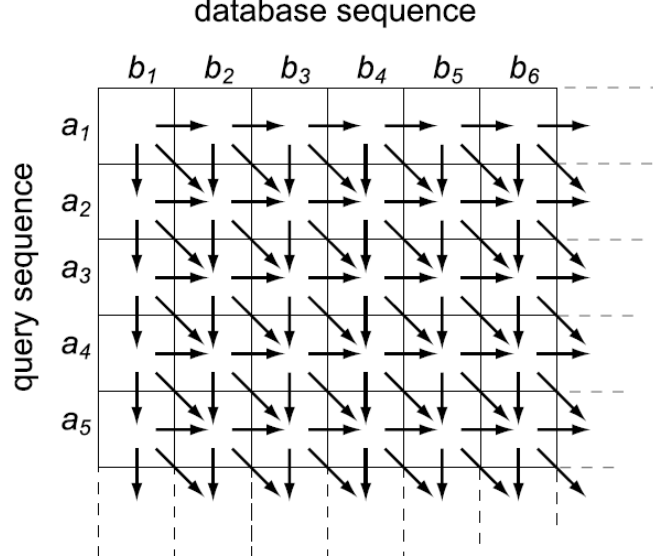
The GPU architecture is built around a fully programmable scalable processor array, organized into a number of streaming multiprocessors (SMs). Depending on the architecture of the GPU, each SM contains a number of scalar processors (SPs) sharing a PBSM (i.e. Per Block Shared Memory). All threads of a thread block are executed concurrently on a single SM. The SM executes threads in small groups, called warps, in an SIMT (i.e. Single Instruction Multiple Threads) fashion [75]. When one thread block is scheduled to execute on an SM, threads in the thread block are split into warps that get scheduled by the SIMT unit. A warp executes one common instruction at a time, but allows for instruction divergence. When divergence occurs, the warp serially executes each branch path. Thus, parallel performance is generally penalized by data-dependent conditional branches and improved if all threads in a warp follow the same execution path. Branch divergence occurs only in a warp, and different warps run independently regardless of common or disjointed code paths being executed [75]. A warp executes one common instruction at a time, therefore all threads in a warp are implicitly synchronized after executing any instruction. Therefore, it is viable to virtualize a warp as an SIMD vector with each thread as a vector element. For the convenience of discussion, hereafter “SIMD vector” will be referred to as “vector”. It must be borne in mind that this is merely a virtualization based on the SIMT model, and hence shares all the features of the SIMT model with an additional ability to conduct vector computations. Simply put, each warp is virtualized as a vector and each thread as an element in that vector.

Figure 5.3 shows the data independences in the alignment matrix. The final value of any cell in the matrix cannot be computed before the values of all cells to the left and above it have been computed. But the calculation of the values of diagonally arranged cells parallel to the minor diagonal are independent and can be done simultaneously in parallel implementation (Figure 5.4 A). This fact has been utilized in Wozniak’s work in 1997 [52]. Rognes and Seeberg [76] later found that using cells along the query sequence was faster despite some data dependences, because loading values along the minor diagonal was too complicated [74]. The advantage of this method was the much-simplified and faster loading of the vector of substitution scores from memory and the disadvantage was that the dependencies within the vector needed to be handled (See Figure 5.4 part B).

Considering the optimal local alignment of a query sequence and a subject sequence as a task, CUD-ASW++ 3.0 implements two approaches for parallel GPU computation: *inter-task* and *intra-task*. In the *inter-task* approach each task is assigned to exactly one thread. Tasks are performed in parallel by differ-



**Figure 5.3:** Data independences in the alignment matrix. The result in the  $(i, j)$ th iteration depend on  $(i - 1, j)$ ,  $(i - 1, j - 1)$  and  $(i, j - 1)$  values.



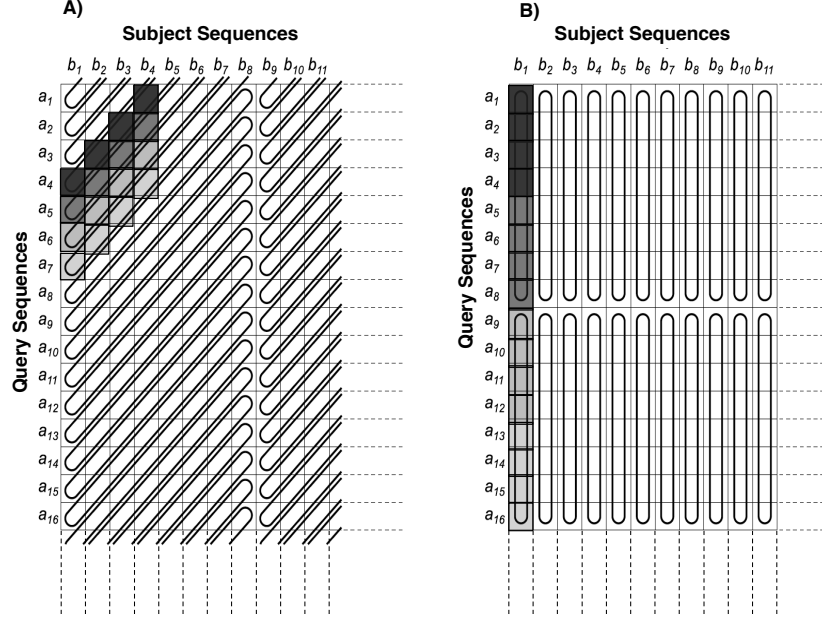
ent threads in a thread block. Inter-task parallelization occupies more device memory but achieves better performance compared to the other approach. In the *intra-task* parallelization each task is assigned to one thread block and all the threads in the thread block cooperate to perform the task in parallel, exploiting the parallel characteristics of cells in the minor diagonals of an alignment matrix (the value of the cells along the minor diagonal in the alignment matrix can be computed in parallel because these calculations are independent). Although slower, intra-task parallelization occupies significantly less device memory and therefore can support longer query/subject sequences.

CUDASW++ 3.0 employs both of the above mentioned stages. A subject sequence length threshold is introduced to separate these two stages (i.e. inter- and intra-task). For subject sequences of length less than or equal to the threshold, the alignments with a query sequence are performed in the first stage (i.e. inter-task) in order to maximize the performance. The alignments of subject sequences of length greater than threshold are carried out in the second stage (i.e. intra-task). Section 5.4.1 further discusses about the subject sequence length threshold and its role in the algorithm.

### Query-sequence profile

Due to the huge number of iterations in the Smith-Waterman algorithm, reducing the number of instructions needed to perform one cell calculation has a significant impact on the execution time. Hence, a simple speed improvement can be achieved by creating a variation of score profile for the query sequence. This profile which can be considered as a query-specific substitution score matrix, is computed only once for the entire search, and will save one memory lookup in the inner loop of the algorithm. The new matrix is indexed by the query sequence position and the database sequence symbol (See Figure 5.9 part A).

**Figure 5.4:** Vector arrangement techniques. (A) Traditional approach with vectors parallel to the minor diagonal. (B) CUDASW++ 3.0 approach with vectors parallel to the query sequence. The first four vectors processed indicated in different shades of gray. For simplicity, vectors of only 4 elements are shown.



In this regard and to follow the Rognes and Seeberg’s virtualized SIMD technique, CUDASW++ 3.0 constructs query profiles, parallel to the query sequence for each possible residue (Figure 5.9 part B). This profile is small enough to be kept in the texture memory.

Given a query  $\mathcal{S}$  defined over an alphabet  $\mathcal{A}$ , a query profile is defined as a numerical string set:

$$P = \{P_r \mid r \in \mathcal{A}\} \quad (5.2)$$

where, for each  $r \in \mathcal{A}$ ,  $P_r$  is a numeric string comprised of substitution scores required for aligning the whole query to the residue  $r$ . CUDASW++ 3.0 constructs the query profile in a way that the  $i^{th}$  value of  $P_r$  is defined as:

$$P_r[i] = sbt(r, \mathcal{S}[i]) \quad \text{where} \quad 1 \leq i \leq |S| \quad (5.3)$$

Inspired by the fact that texture instructions output filtered samples, typically a four-component (RGBA) colour [77], the implemented query profile is re-organized using a packed data format, where each numerical string,  $P_r$ , is packed and represented using the *char4* vector datatype. In this way, four substitution scores

are realized using only one texture fetch, thus significantly improving texture memory throughput. Like the query profile, each subject sequence is also re-organized using a packed data format, where four successive residues of each subject sequence are packed together and represented using the *uchar4* vector data type.

## 5.4 MR-CUDASW

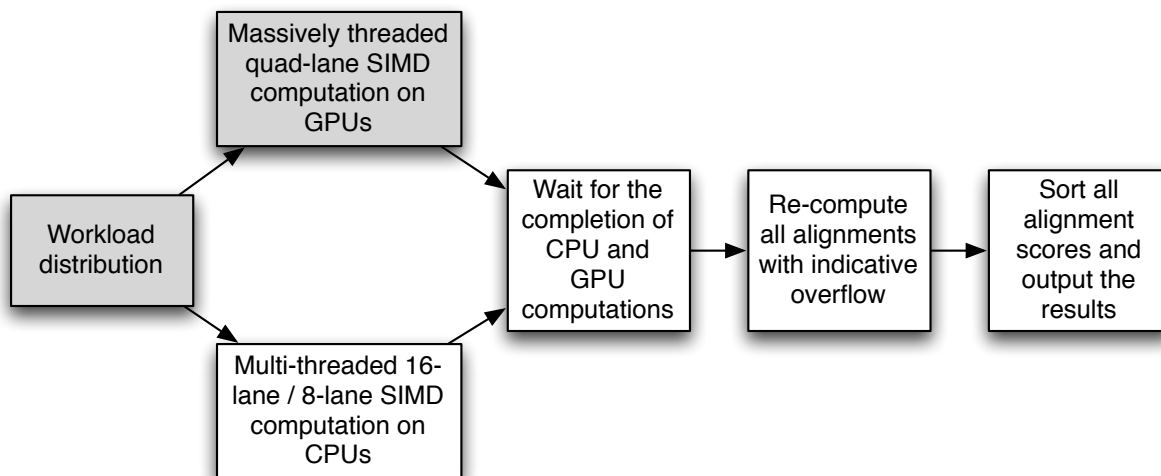
As apparent by the results from the evaluation outlined in Section 6.1.2, CUDASW++ 3.0 shows the best performance among the benchmarked GPU-based sequences aligners. However, considering that the elapsed real time of the whole MIRA assembly process is significantly smaller than the elapsed real time of CUDASW++ 3.0 (Tables 6.1 and 6.2), the performance of this software needs to be improved further in order to replace MIRA’s overlap determination technique.

It is important to remember that unlike most of the currently existing GPU-based implementations of the Smith-Waterman algorithm that are designed to perform protein database searches, this study is an application where nucleotide sequences are aligned. As mentioned before, this study assumes that its target set of sequences has the characteristics of reads from a medium-length next generation sequencing technology (e.g. Ion Torrent). It also assumes that these reads are already quality-filtered and are ready for downstream analysis (e.g. assembly). Considering the above assumptions, it is expected that the lengths of these sequences vary from  $\sim 100\text{bp}$  to  $\sim 300\text{bp}$ . This range of sequence lengths leads to a far smaller sequence length deviation compared to the case where a query sequence is aligned with all the others sequences residing in a database. In this regard, highlighted stages in Figure 5.5, which shows the program workflow of CUDASW++ 3.0, were replaced by custom methods specifically designed to handle medium-length reads. Eventually, all these modifications were combined together to create MR-CUDASW: the final improved version of CUDASW++ 3.0, specifically designed to handle medium-length sequences. The following sections describe why these stages were selected for further modifications and what improvements have been made.

### 5.4.1 Sequence length deviation & thread scheduling

As discussed above, CUDASW++ 3.0 and its predecessors (i.e. CUDASW++ 1.0 and 2.0) use two stages to complete the database searches: the first stage uses inter-task parallelization employing thread-level parallelism, and the second stage utilizes intra-task parallelization using data parallelism. As described in Section 5.3.1, CUDASW++ 3.0 makes use of a subject sequence length threshold to separate the two stages. For subject sequences of length less than or equal to the threshold, the alignments with a query sequence are performed in the first stage in order to maximize the performance. Although this threshold is highly dependent on the memory size of the graphics hardware in use, CUDASW++ 3.0 has the threshold hardcoded and set to 3,072. Considering the relatively short lengths of our target reads and the expanded per-thread local memory size of the Kepler architecture, our improved version of CUDASW++ 3.0 only implements the inter-task parallelization approach. It has already been proven by Liu et al. [75] that the inter-task

**Figure 5.5:** Program workflow of CUDASW++ 3.0. The highlighted boxes were subject to modifications.

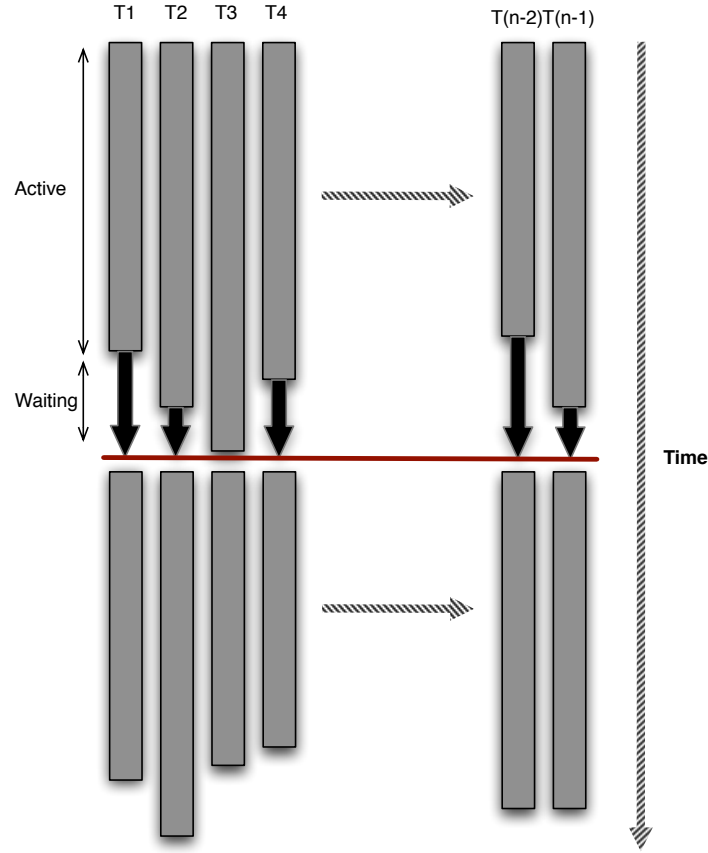


parallelization dominates the total runtime. Considering that each task (i.e. finding the optimal alignment between two sequences) is assigned to exactly one thread, improving the existing thread scheduling technique was a viable option.

The length of biological sequences residing in a dataset varies. Since the size of the required memory and the computation time of a thread is determined by the longest sequences, the length distribution of sequences influences the application's performance. As a result, long lags are introduced for the alignment of the shorter sequences. In order to achieve high efficiency for inter-task parallelization, the runtime of all threads in a thread block should be roughly identical (i.e. threads must be synchronized). Thread synchronization in CUDA follows strict synchronization rules: All threads in a block must hit the synchronization point or none of them must hit synchronization point [78] (Figure 5.6). Hence, it is important that for two adjacent threads in a thread block, the difference between the products of the lengths of the associated sequences be minimized. An easy, yet effective, solution is to sort the sequences in the database and then partition them into number of groups each containing sequences with roughly identical size. This technique has been implemented in CUDASW++ algorithm since the earliest version of this program [18] and it has been shown that a lot of superfluous computations can be omitted as the result of this technique. See Figure 5.7.

Sorting the sequences in an order of sequence length simplifies thread scheduling and reduces the size of buffers, which in turn leads to decreasing the number of superfluous computations. However, it can be beneficial to distribute the workload in a more intuitive manner. As stated above, the sequence length deviation generally causes runtime imbalance between threads and consequently wastes GPU compute power. In this regard, instead of distributing the workload only according to the compute power of CPUs and GPUs, the technique employed in CUDASW++ 3.0, MR-CUDASW also uses the length of the query sequence, the

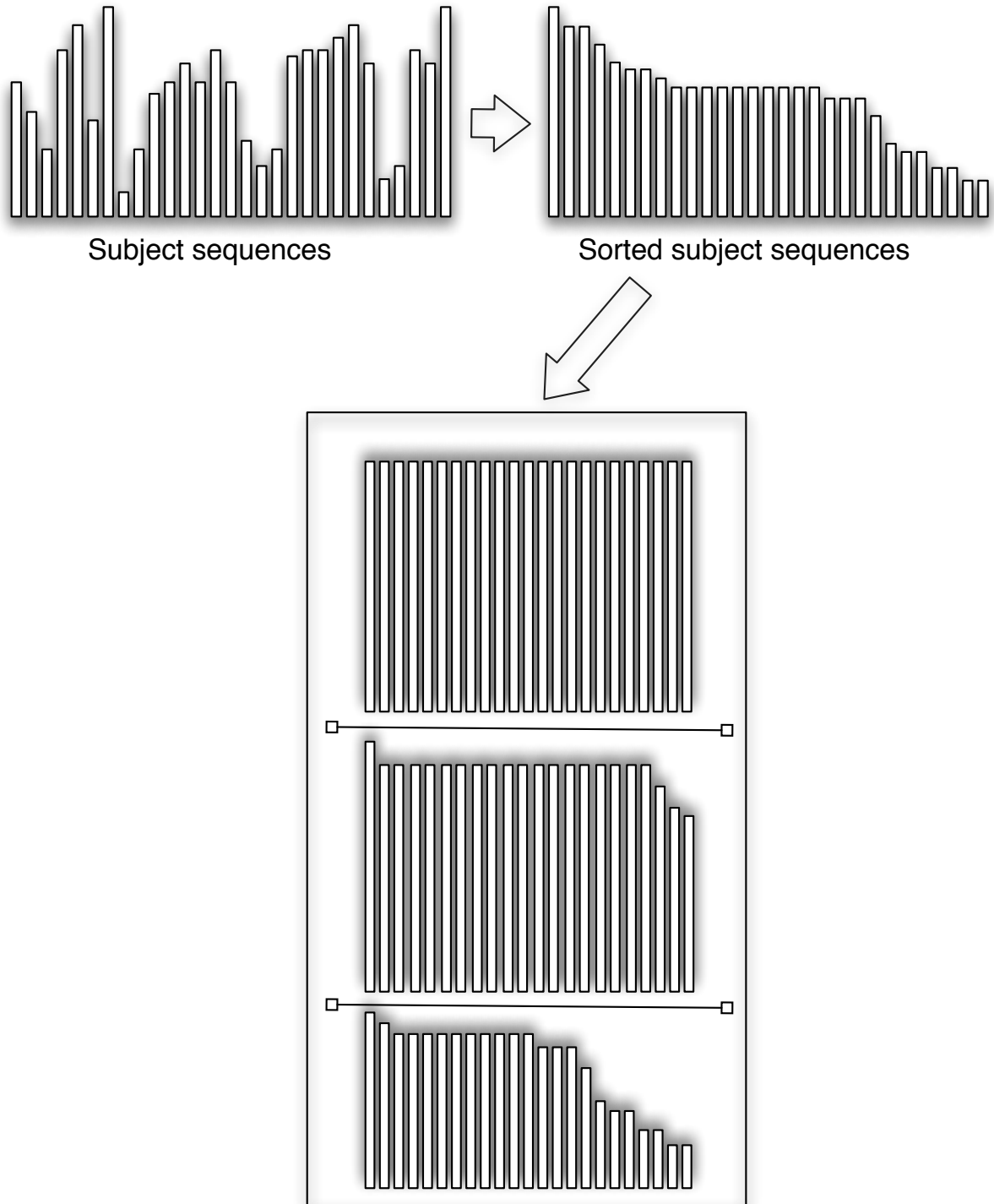
**Figure 5.6:** CUDA thread synchronization rules. All threads in a block must hit the synchronization point (red line in the figure) or none of them must hit synchronization point.



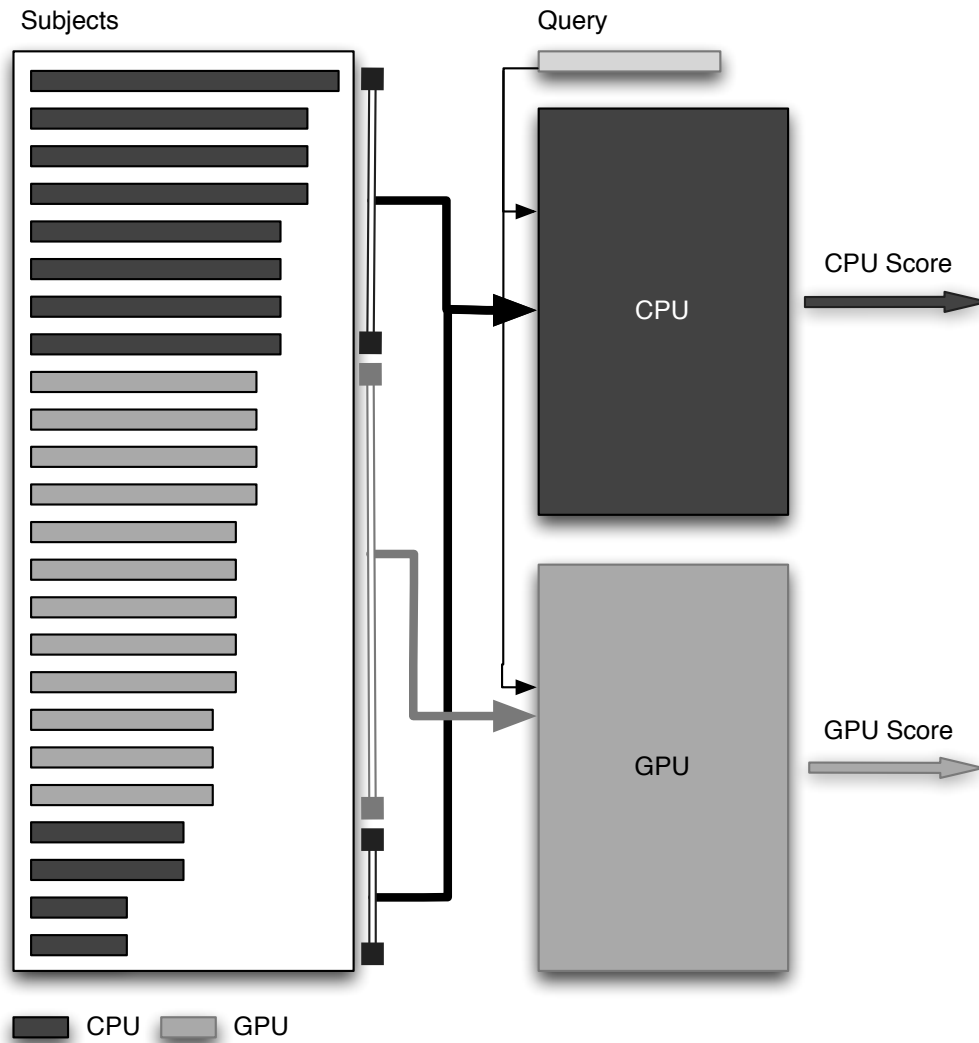
length of the subject sequence, and the average sequence length deviation calculated for all the sequences residing in a dataset. In this technique, if the difference between the query and subject sequences is larger than the average sequence length deviation of the dataset, the subject sequence will be assigned to the CPU. Referring back to Equation 5.1, the number of sequences distributed to the CPU will not be larger than  $(1 - R)$  times the total number of residues in the dataset. Therefore, the workload is distributed based on the length of the query sequence, the length of the subject sequence, and the average sequence length deviation calculated for the dataset, as well as the compute power of the CPUs and GPUs. See Figure 5.8.

To evaluate the effects of the above mentioned modification, the modified software was evaluated against the original algorithm. The performance of EMBOSS water was also provided as a point of comparison. To ensure the fidelity of the alignment scores in each step, a method analogous to the procedure outlined in Section 5.3.1 was employed. All the comparisons were carried out using simulated and real WGS data (Table 5.1 describes the characteristics of the data). The results of these evaluation are presented in Section 6.2.1.

**Figure 5.7:** Arrangement of subject sequences in the database for the inter-task parallelization. To achieve high efficiency for inter-task parallelization, all the sequences in the database are sorted and then partitioned into number of groups each containing sequences with roughly identical size.



**Figure 5.8:** The workload distribution technique employed in MR-CUDASW. This figure demonstrates the arrangement of the subject sequences in the database. All the sequences in the database are sorted and partitioned into a number of groups each containing sequences with roughly identical size. A query sequence will be selected from each batch. If the difference between the query and subject sequences is larger than the average sequence length deviation of the dataset, the subject sequence will be assigned to CPU.



### 5.4.2 Query profile

The new workload distribution technique leads to a significant improvement in the performance of the software (Section 6.2.1). However, referring to Tables 6.1 and 6.2, the elapsed real time of the entire MIRA process is still notably smaller than the time that MR-CUDASW takes to determine overlaps between pairs of medium-length sequences. More improvements are needed.

As already indicated, CUDASW++ 3.0 employed the sequential layout query profile [76] in which the SIMD registers contained values parallel to the query sequence (Figure 5.4, part B). A disadvantage introduced by processing the values vertically is that conditional branches are placed in the inner loop to compute  $H$  (See Algorithm 1). With conditional code the execution time is dependent on the length of the query string and the database, the scoring matrix and gap penalties [50].

In order to address these shortcomings, another implementation of the Smith-Waterman algorithm where the SIMD registers are parallel to the query sequence, but are accessed in a striped pattern, was introduced by Farrar et al. [50]. The striped Smith-Waterman implementation takes a similar approach to the Rognes and Seeberg's technique [76] by pre-calculating the query profile, but with a different layout than used in that method (Figure. 5.9).

When calculating  $H[i, j]$  the value from the scoring matrix  $W(seqA[i], seqB[j])$  is added to  $H[i1, j1]$ . To avoid the lookup of  $W(seqA[i], seqB[j])$  for each cell, Rognes and Seeberg [76] calculated a query profile parallel to the query for each possible residue. The query profile is calculated just once for each database search. Then the calculation of  $H[i, j]$  requires just an addition of the pre-calculated score to the previous  $H[i, j]$ . The striped Smith-Waterman implementation takes a similar approach by pre-calculating the query profile, but with a different layout than Rognes.

The layout used by the query profile is a striped access parallel to the query sequence,  $\mathcal{S}$ . The query is divided into equal length segments,  $s$ . The number of segments,  $p$ , is equal to the number of elements being processed in the SIMD register. The length of each segment,  $t$ , is  $(|\mathcal{S}| + p - 1)/p$ . If the query is not long enough to fill all the segments,  $t > |\mathcal{S}|$ , the segments are padded with null entries. The query residues are represented by  $q$ . The query segments are defined as follows. In the following formula,  $n$  index represents the segment number

$$s_n = q_{t(n-1)+1}, q_{t(n-1)+2}, \dots, q_{t(n-1)+t} \quad \text{where } 1 \leq n \leq |s|.$$

The length of each segment is defined as:

$$|s_n| = t = (|\mathcal{S}| + p - 1)/p. \quad (5.4)$$

As mentioned earlier, if  $|\mathcal{S}|$  is not a multiple of  $p$ ,  $\mathcal{S}$  is first padded with dummy residues. For the sake of simplicity, herein it is assumed that  $|\mathcal{S}|$  is a multiple of  $p$ . Given a query  $\mathcal{S}$  defined over an alphabet  $\mathcal{A}$ , a query profile is defined as a numerical string set:

$$P = \{P_r \mid r \in \mathcal{A}\}.$$



Accordingly, each numerical string  $P_r$  of a query profile can be considered as a set of non-overlapping, consecutive  $p$ -length vector segments of  $|s^n|$  elements. Hence  $P_r$  in the new approach is defined as:

$$P = \{P_r \mid r \in \mathcal{A}\} P_r[i] = sbt(r, \mathcal{S}[(i-1)\%p] \times |s_x| + (i-1)/p + 1]) \quad \text{where } 1 \leq i \leq |S| \quad (5.5)$$

**Figure 5.9:** Approaches to vectorization of Smith-Waterman alignments. (A) Vectors along the query, described by Rognes and Seeberg [74] (used in CUDASW++ 3.0). (B) Striped approach, described by Farrar [69] where  $p = 4$  and  $|s| = t = 4$ . The row encircled by the dash line illustrates a query profile ( $P_r$ ).

		Query Sequences $\mathbf{S}$															
		$q_1$	$q_2$	$q_3$	$q_4$	$q_5$	$q_6$	$q_7$	$q_8$	$q_9$	$q_{10}$	$q_{11}$	$q_{12}$	$q_{13}$	$q_{14}$	$q_{15}$	$q_{16}$
Residues in $\mathbf{A}$	<b>A</b>	-1	-1	5	1	0	-1	-1	1	-2	-1	0	1	0	-1	-1	-1
	<b>C</b>	-2	-3	-1	-1	-3	-3	-2	-1	-2	-3	-1	-1	-1	-3	-2	-3
	<b>G</b>	-2	4	-1	5	-2	1	-2	5	-3	-2	-3	0	-1	4	-2	4
	$\vdots$		...						...						...		
	<b>N</b>	-1	4	-1	0	-2	1	-2	0	-3	4	-3	0	-1	4	0	4

(A)

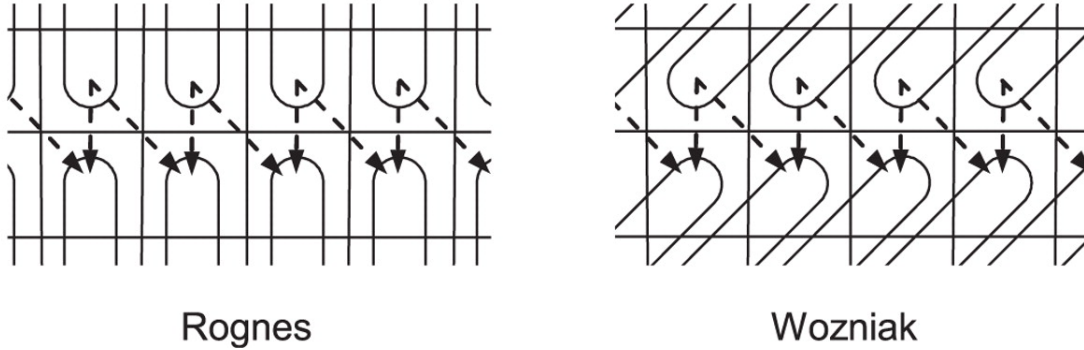
		Query Sequences $\mathbf{S}$															
		$s_1$				$s_2$				$s_3$				$s_4 = s_p$			
		$q_1$	$q_{2t+1}$	$q_{3t+1}$	$q_{4t+1}$	$q_2$	$q_{2t+2}$	$q_{3t+2}$	$q_{4t+2}$	$q_3$	$q_{2t+3}$	$q_{3t+3}$	$q_{4t+3}$	$q_4$	$q_{2t+4}$	$q_{3t+4}$	$q_{4t+4}$
Residues in $\mathbf{A}$	<b>A</b>	-1	0	-2	0	-1	-1	-1	-1	5	-1	0	-1	1	1	1	-1
	<b>C</b>	-2	-3	-2	-1	-3	-3	-3	-3	-1	-2	-1	-2	-1	-1	-1	-3
	<b>G</b>	-2	-2	-3	-1	4	1	-2	4	-1	-2	-3	-2	5	5	0	4
	$\vdots$		...						...						...		
	<b>N</b>	-1	-2	-3	-1	4	1	-3	4	-1	-2	-3	0	0	0	0	4

(B)

Each element of the SIMD registers maps to one segment. The first element in the vector maps to  $s_1$ , the second element in the vector maps to  $s_2$ , till the last element in the vector maps to  $s_p$ . The vectors move uniformly across the segments, so SIMD registers process the  $i^{th}$  element of all the segments. Both the Wozniak [52] and Rognes and Seeberg [76] implementations have data dependencies between the previous  $H$  vector and the current  $H$  vector (Figure 5.10). Before  $H$  is calculated, the last element in the previous vector is moved to the first element in the current vector. By using the striped query access, these data

dependencies are moved out of the inner loop and processed just once in the outer loop when processing the next database residue. This technique, already implemented by Liu et al. [75] in CUDASW++ 2.0 (replaced in CUDASW++ 3.0 with Rognes and Seeberg’s [76] technique), is cited to achieve the best performance [74]. The performance of this technique is highly dependent on query length though [54].

**Figure 5.10:** Data dependencies between SIMD registers holding the H values with the Rognes and Wozniak implementations. The last element in the previous vector is inserted in the current vector when calculating the next H vectors. Taken from a study by Micheal Farrar [69].



Considering the length of this study’s target sequences and the sequence length deviation of the dataset, Farrar’s approach [69] was selected as the alternative method of constructing sequence query profiles. The CUDASW++ 2.0 implementation of Farrar’s method of query profile construction employs global memory for storing intermediate elements. Our technique, in contrast, stores the striped query profile in texture memory to exploit the texture cache. Subject sequences and the query profile are also stored using the scalar data type in an unpacked fashion because the inner loop is a *for* loop without manual unrolling<sup>4</sup>. Considering that the Kepler architecture has 512 KB per-thread local memory, the intermediate element values of  $H(i, j)$  are stored in local memory.

### 5.4.3 Ensuring the fidelity of the result

As presented above, a number of improvements and changes have been made to the original implementation of CUDASW++ 3.0 (v3.0.14). Workload distribution, query profile construction, and thread scheduling techniques implemented in CUDASW++ 3.0 were replaced by custom methods specifically designed to handle medium-length reads. To ensure the correctness of the resultant tool, its results are evaluated against the EMBOSS sequential implementation of Smith-Waterman algorithm. All the resultant scores from two programs are then compared using a custom shell script and in all the cases scores were identical (data not shown).

<sup>4</sup>A loop transformation technique that attempts to optimize a program’s execution speed at the expense of its binary size (space-time tradeoff). The transformation can be undertaken manually by the programmer or by an optimizing compiler.

## CHAPTER 6

### RESULTS

This chapter of the thesis presents results concerning the comparison of some of the popular GPU-accelerated sequence alignment software based on their performance. The goal of this study was to determine their suitability for the purpose of this study (Section 6.1). Section 6.1.1 describes the metrics used for this evaluation and finally Section 6.1.2 presents the results of the comparison. Results of the attempts to improve the fastest existing GPU-based implementation of the Smith-Waterman algorithm is presented in Section 6.2. By comparing the modified version of the software to the original algorithm, this section, step by step, presents the effects of our modifications on the performance of the original software. Finally, Section 6.3 evaluates the efficacy of the resultant algorithm (i.e MR-CUDASW) when compared to an OLC-based assembler.

#### 6.1 Benchmarking GPU-accelerated Smith-Waterman tools

As stated before, the goal of this thesis is to determine the possibility of improving the performance of the overlap determination stage of an OLC-based assembler by using a GPU-based implementation of the Smith-Waterman algorithm. In this regard, identifying a set of GPU-accelerated aligners performing the Smith-Waterman algorithm was needed. Referring back to section 5.2.1, there are number of sequence alignment software tools that focus on implementing the Smith-Waterman algorithms on GPUs. Our brief review of some of the more popular software, based on their published description, assisted us to determine their suitability for the purpose of this study and narrow down the list of the algorithms that could potentially be useful. However, the published results presenting the performance of these software tools can be confusing. These results depend heavily on the length of sequences and the hardware used to perform the comparison. Hence more in-depth examination of each software tool is required. To this end, three simulated nucleotide datasets as well as three real nucleotide datasets containing 1000, 10,000, and 100,000 reads were used to benchmark these algorithms.

### 6.1.1 Metric

To measure the performance of the various implementations of Smith-Waterman algorithms, the Giga Cell Updates per Second (GCUPS) metric is commonly used. GCUPS represents the number of updates of the similarity matrix per unit time. Equation 6.1 formulates the relation, where  $R$  denotes the total number of residues in the dataset, and  $T$  denotes the time it takes to perform the computation in seconds.

$$GCUPS = \frac{R^2}{T \times 10^9} \quad (6.1)$$

### 6.1.2 Benchmarking

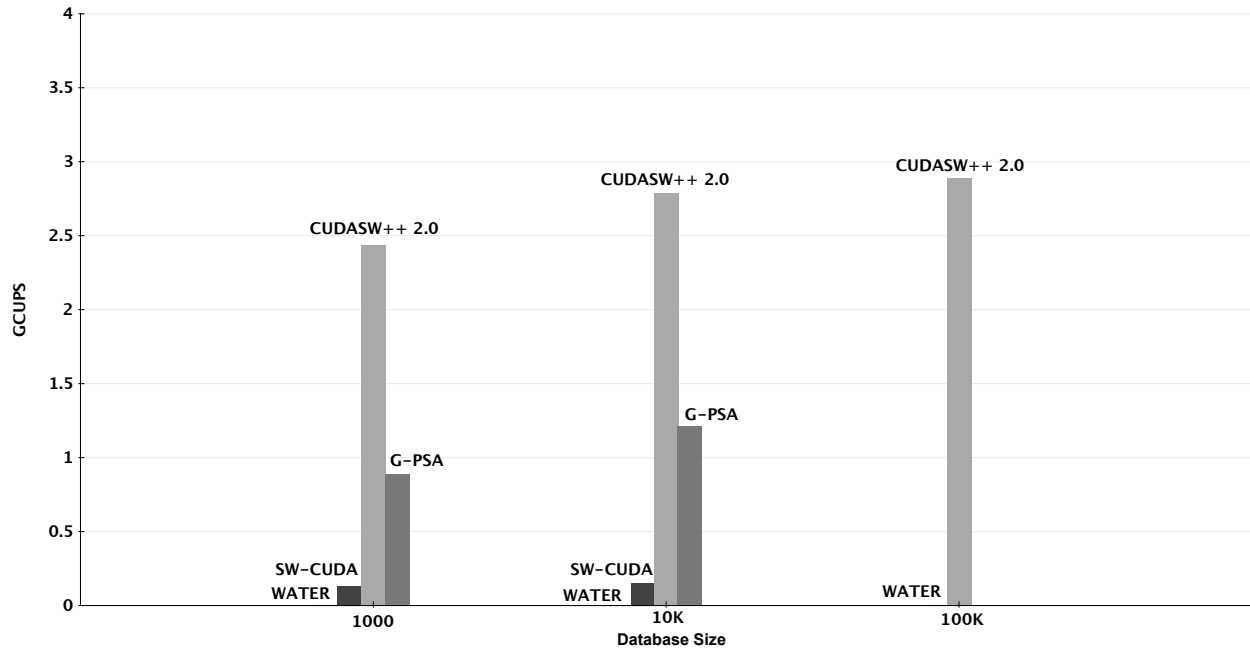
This section describes the result of the evaluation of the performance of the following algorithms: CUDASW++ 3.0 (v3.0.14), CUDASW++ 2.0 (v2.0.10), G-PAS(v2.0), SW-CUDA (v1.92) and EMBOSS water (v6.4.0.0). Except for CUDASW++ 3.0 which is explicitly designed to run on the Kepler architecture and water, all the above mentioned software tools are designed to take advantage of many-core parallelism on GPUs with Fermi architecture and cannot be readily transformed onto a machine with Kepler architecture GPU. Hence our comparison was carried out in two steps. First, all the software tools designed based on the Fermi architecture were compared against each other and water on a Lenovo M91p Tower with Intel Core i7-2600 Quad Core (3.40/3.80GHz, 8MB Intel Smart Cache, 1333MHz FSB), 4GB RAM and NVIDIA GeForce GT 640 GPU (901MHz Engine Clock, 1782MHz, GDDR3 2GB, 128-bit). All the participating software in this evaluation exploited an opening gap penalty of 10 and extending gap penalty of 2.

To ensure the correctness of the results, all the alignments scores were evaluated against the EMBOSS sequential implementation of Smith-Waterman algorithm. All the scores from two programs were compared using a custom shell script. All the participating algorithms reported identical alignment scores to the scores reported by water (data not shown).

After confirming the fidelity of the outcome alignment scores, all data was copied to a fast local disk to reduce the influence of the computer networks and minimize file reading time. The output from all programs was discarded to minimize performance differences due to the amount of output. To simplify the comparison, all the sequences aligners were run one at a time (without any other program running using the machine) using their default parameters. All programs were run 10 times and the median total elapsed execution time was recorded. Figures 6.1 and 6.2 illustrate the results of this evaluation. As it is evident in the following figures, water achieved nearly constant, very poor performance ( $< 0.01GCUPS$ ) for all the database sizes in our evaluations. Performance details of the algorithms participating in this evaluation, including the recorded elapsed real time, user time, and system time, can be found in Appendix C.

The fastest sequence aligner running on a Fermi architecture GPU was then modified to run on a Kepler architecture. Considering that our Kepler architecture GPU is installed in a Mac Pro machine, additional modifications were required for CUDASW++ 2.0 to participate in this evaluation (some data types and

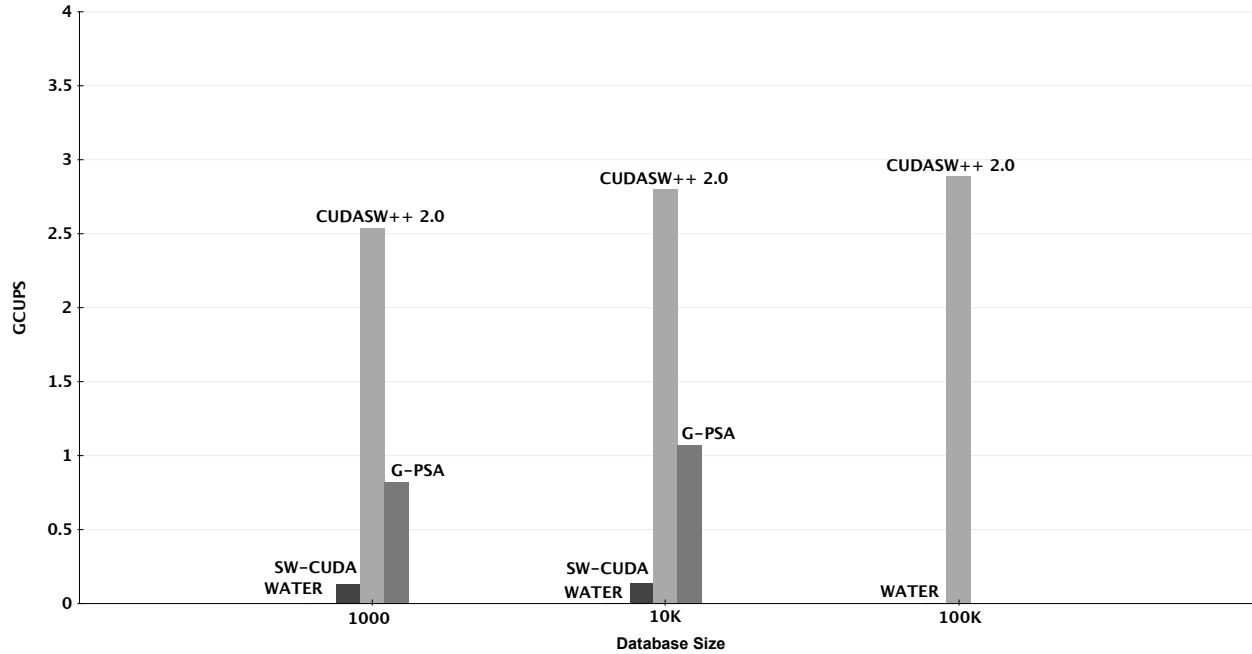
**Figure 6.1:** Performance details of GPU-accelerated alignment software tools determining overlaps between reads in simulated datasets of varying sizes on GT 640 Fermi architecture GPU. G-PSA and SW-CUDA failed to finish the alignment process within a reasonable time and hence, there is no information to report their performances.



libraries were the subject of modifications). This algorithm was compared to CUDASW++ 3.0 (v3.0.14) and EMBOSS water (v6.4.0.0). All the comparisons were carried out on a single NVIDIA GeForce GTX 680 graphic card with 30 SMs (Streaming Multiprocessors) comprises 192 CUDA SP (Scalar Processor) cores sharing a configurable 64 KB on-chip memory and 2.048 GB RAM installed in a Mac Pro with a 2.66 Quad core Intel Xeon CPU (256 KB L2 Cache, and 8 MB L3 Cache) and 16 GB RAM. Again, all the resultant alignment scores were compared against EMBOSS water alignment scores and in all the cases scores were identical (data not shown). Like the first phase, all the outputs from programs were redirected to `/dev/null` to minimize performance differences. Except for the same open and gap penalties and the same scoring matrix, all the programs utilized their default parameters. All the programs were run 10 times and the median total elapsed execution time was recorded. Figures 6.3 and 6.4 show the performance details of GPU-accelerated alignment software aligning datasets of various size on a Kepler architecture GPU.

To answer the question of whether or not it is possible to improve the performance of an OLC-based assembly software by using a GPU-based acceleration techniques, we need to compare the performance of the overlap determination stage of an OLC-based assembler (e.g. MIRA) with the the fastest GPU-based implementation of Smith-Waterman algorithm. Tables 6.1 and 6.2 illustrate the performance of CUDASW++ 3.0 and MIRA for simulated and real WGS datasets with varying number of queries. It should be noted that the numbers reported in order to present the performance of the MIRA overlap determination technique, in fact show the performance of the whole assembly process. The MIRA's overlap determination stage could not

**Figure 6.2:** Performance details of GPU-accelerated alignment software tools determining overlaps between reads in real WGS datasets of varying sizes on GT 640 Fermi architecture GPU. G-PSA and SW-CUDA failed to finish the alignment process within a reasonable time and hence, there is no information to report their performances.



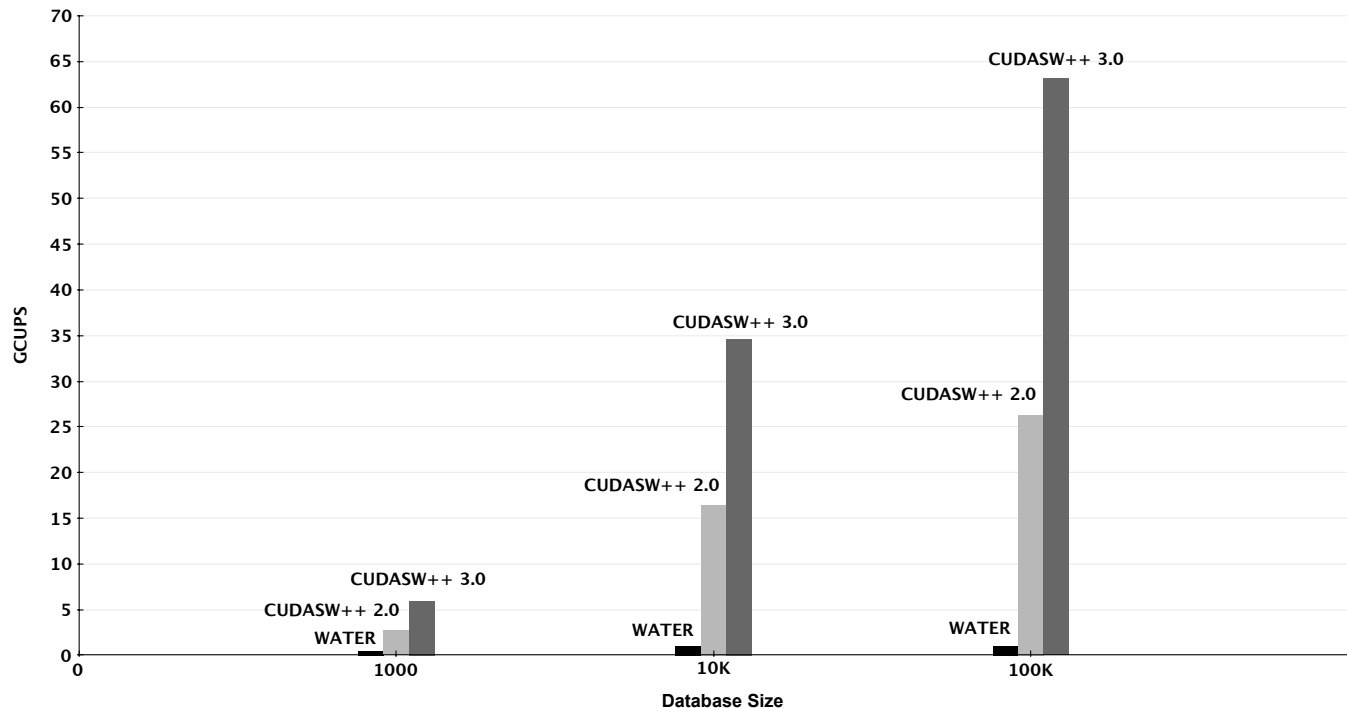
be extracted from MIRA package or individually implemented due to lack of documentation and inadequate modularity in the software design. Nevertheless, our evaluation results resolved the need for reimplementing of this stage by illustrating the poor performance of CUDASW++ 3.0. This evaluation provides insights into the relative importance of improving the performance of CUDASW++ 3.0 as the fastest existing SW implementation for general-purpose GPUs (GP-GPUs).

## 6.2 Improving CUDASW++ 3.0

As the result of the evaluation outlined in Section 6.1.2, CUDASW++ 3.0 shows the best performance among the other benchmarked GPU-based implementations of the Smith-Waterman algorithm. However, considering that, for the datasets of considerable sizes ( $> 10,000$  queries), the elapsed real time of the entire MIRA's assembly process is significantly smaller than the elapsed real time of CUDASW++ 3.0 (Tables 6.1 and 6.2), the performance of this software still needs to be improved in order to replace MIRA's overlap determination technique.

To this end, CUDASW++ 3.0 was closely analyzed and the technical features that make this software far faster than the other benchmarked software were identified. Some of these techniques were further expanded to improve the performance of this software tool and some were replaced. Over the course of this phase, workload distribution, query profile construction, and thread scheduling techniques implemented in

**Figure 6.3:** Performance details of GPU-accelerated alignment software tools determining overlaps between reads in simulated datasets of varying sizes on GTX 680 Kepler architecture GPU.



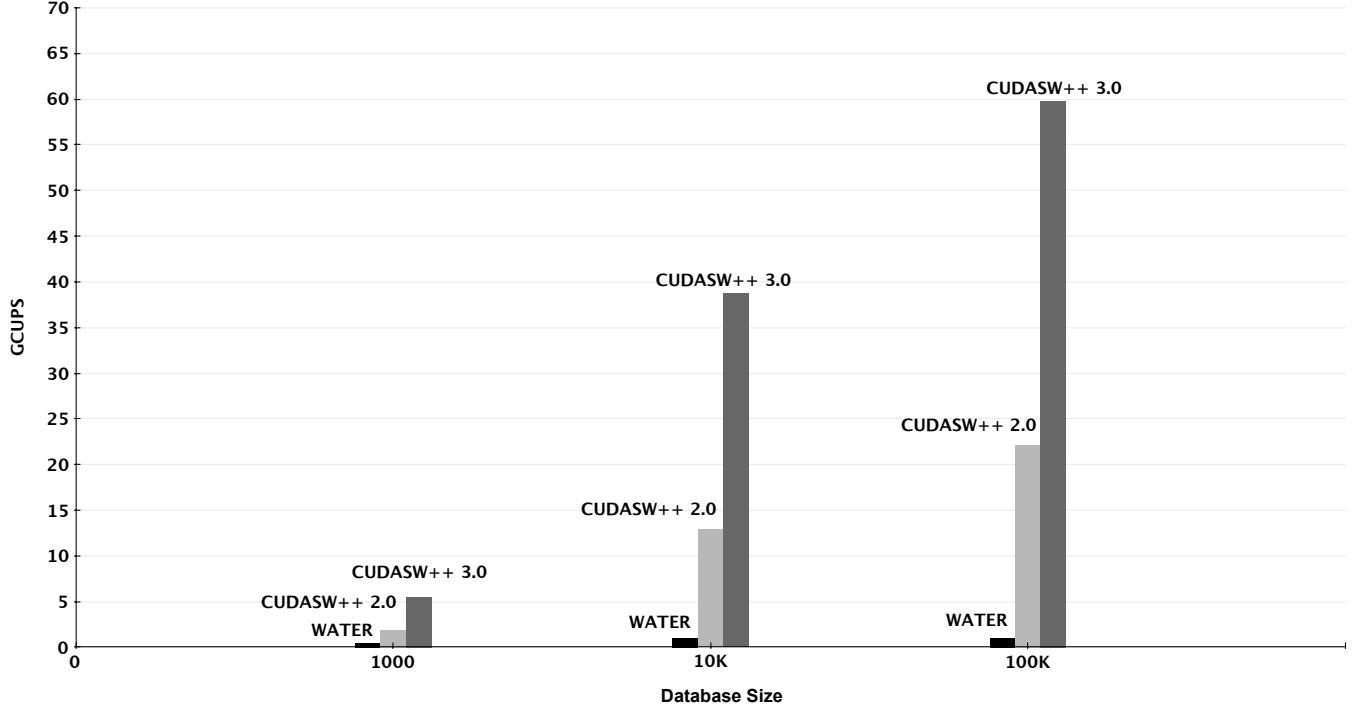
CUDASW++ 3.0 were replaced by techniques specifically customized to handle medium-length reads (See Section 5.3).

As discussed in Section 5.3, the modifications were implemented step by step, leading to a final improved version of the original programs. To better illustrate the effects of each step, the modified version of the software was compared to the original algorithm after each modification. The performance of EMBOSS water is also provided as point of comparison. To ensure the fidelity of the alignment scores in each step, a method analogous to the procedure outlined in Section 5.3.1 was employed; the resultant software was evaluated against the EMBOSS sequential implementation of Smith-Waterman algorithm. All the resultant alignment scores from the modified version of the software were compared against EMBOSS water alignment scores using a custom shell script. The following section briefly reviews the process of replacing and customizing the thread scheduling and workload distribution techniques used by CUDASW++ 3.0 and presents the performance data.

### 6.2.1 Sequence length deviation and thread scheduling

As outlined in detail in Section 5.4, various modifications were made to the first and second stages of the CUDASW++ 3.0 algorithm in order to customize the original algorithm to be more compatible with medium-length nucleotide sequences. Considering the specific characteristics of this study's target sequences (i.e. medium-length of the query sequences and the small overall sequence length deviation), CUDASW++

**Figure 6.4:** Performance details of GPU-accelerated alignment software tools determining overlaps between reads in real WGS datasets of varying sizes on GTX 680 Kepler architecture GPU.



3.0's thread scheduling and workload distribution techniques were subject to modifications (Section 5.4.1). By focusing on a distinct range of sequence lengths and taking advantage of the expanded per-thread local memory size of the Kepler architecture, MR-CUDASW resolved the need of implementing the intra-task parallelization method. In addition, instead of distributing the workload only according to the compute power of CPU and GPUs (technique employed in CUDASW++ 3.0), the workload distribution is distributed based on the length of the query sequence, the length of the subject sequence, and the total sequence length deviation calculated for the dataset as well as the compute power of the CPU and the GPUs.

Using the mentioned thread scheduling and workload distribution techniques, the waste of CPU and GPU SIMD instructions are avoided as all alignments in all lanes are completed at the same time. It can also avoid the computational imbalance between threads within a warp and a thread block. Figures 6.5 and 6.6 illustrate the significant effects of these modifications on improving the performance of the modified software.

### 6.2.2 Query profile

As illustrated in Figures 6.5 and 6.6, modifying CUDASW++ 3.0's thread scheduling and workload distribution techniques led to a significant speedup. However, as shown in Tables 6.1 and 6.2, and Figures 6.5 and 6.6, the performance of the whole MIRA's assembly process is still far better than the performance of the modified version of the algorithm. More improvements were needed.

Referring back to Section 5.4.2, during the computations of the matrix cells the values in the two arrays



**Table 6.1:** Performance details of CUDASW++ 3.0 and MIRA. Datasets of various number of simulated data were used for this experiment.

Software	Elapsed real time <sup>a</sup>	User time <sup>b</sup>	System time <sup>c</sup>	GCUPS
SYN_1000.fna				
CUDASW++ 3.0	0 m 7 s	0 m 9 s	0 m 1 s	5.95
MIRA 4.0	0 m 26 s	0 m 3 s	0 m 4 s	1.50
SYN_10K.fna				
CUDASW++ 3.0	1 m 53 s	6 m 20 s	0 m 11 s	34.64
MIRA 4.0	0 m 58 s	0 m 12 s	0 m 11 s	67.49
SYN_100K.fna				
CUDASW++ 3.0	102 m 32 s	408 m 18 s	2 m 16 s	63.15
MIRA 4.0	37 m 6 s	7 m 11 s	2 m 33 s	174.53

<sup>a</sup>The time or difference between a beginning time and an ending time of the program.

<sup>b</sup>Total amount of CPU-time that the process spent in user mode.

<sup>c</sup>Total amount of CPU-time that the process spent in kernel mode.

with the  $H$  and  $E$  values usually have to be read and written once for each matrix cell. These arrays are usually small enough to be cached at a close cache level, so the memory access time should not be a major concern. However these data still need to be written and read back for each cell. Since there is ample space for keeping the  $H$ ,  $E$  and  $F$  values of a few cells in the registers, it is possible to reduce running time somewhat by computing a few consecutive cells along the database sequences before moving on to the next query residue. To this end, CUDASW++ 3.0 makes use of the Rognes and Seeberg’s technique to construct query profiles, parallel to the query sequence for each possible residue (Figure 5.4 part B). In this way, four substitution scores are realized using only one texture fetch, thus significantly improve the texture memory throughput. Although effective, the execution time of this method is heavily dependent on the length of the query string and the database, the scoring matrix and gap penalties. As discussed in Section 5.4.2, the striped implementation of the Smith-Waterman algorithm introduced by Farrar et al. [50] was an alternative to address the shortcomings of Rognes and Seeberg’s technique. This approach is believed to be the fastest implementation of this algorithm, though highly dependent on query length [74]. The effects of this technique were also explored in the earlier version of CUDASW++ 3.0 [75].

Once again, a distinct range of the target sequences provides this study with the opportunity for exploring the effects of the striped implementation of the Smith-Waterman algorithm on a Kepler architecture GPU, without being concerned about this technique’s high dependency on query length. Herein, by improving and customizing the already-implemented striped query profile technique for the Kepler architecture and taking advantage of this architecture’s expanded texture memory, the new modified program illustrated superior speedup comparing to the original version of the software. Figures 6.7 and 6.8 show the effects of this modification on improving the performance of CUDASW++ 3.0.

**Table 6.2:** Performance details of CUDASW++ 3.0 and MIRA. Datasets of various number of real WGS data were used for this experiment.

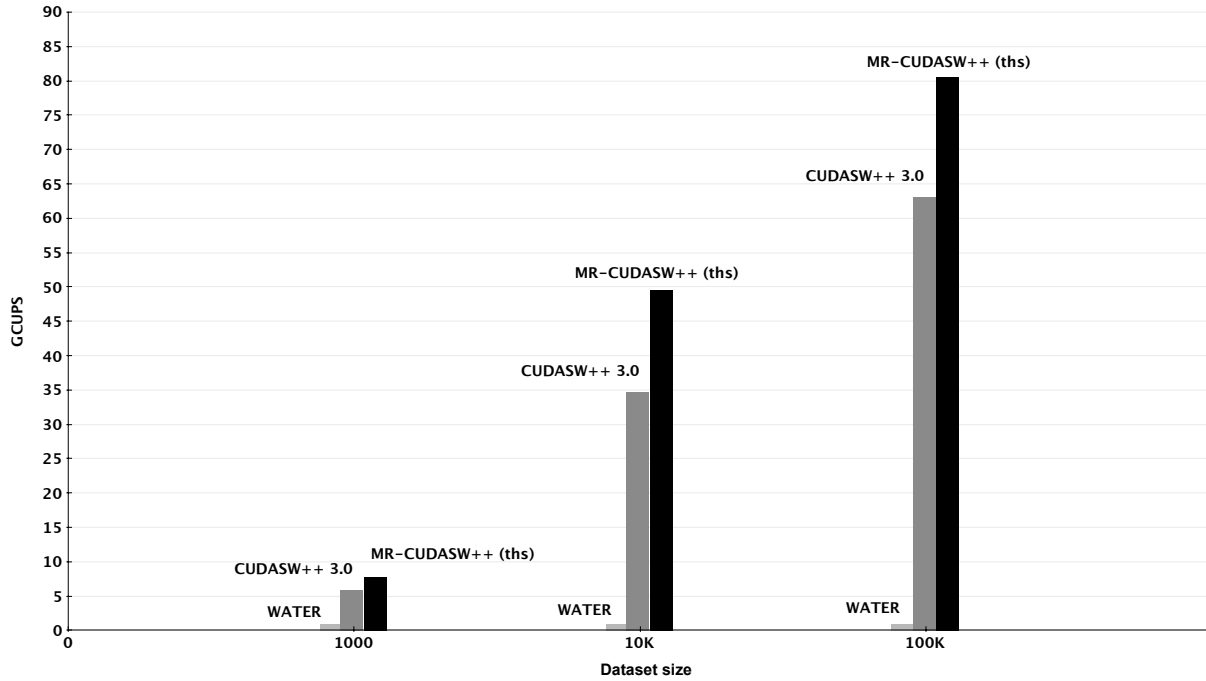
Software	Elapsed real time <sup>a</sup>	User time <sup>b</sup>	System time <sup>c</sup>	GCUPS
ENV_1000.fna				
CUDASW++ 3.0	0 m 6 s	0 m 8 s	0 m 1 s	5.51
MIRA 4.0	0 m 12 s	0 m 0 s	0 m 1 s	2.75
ENV_10K.fna				
CUDASW++ 3.0	1 m 33 s	5 m 17 s	0 m 10 s	38.72
MIRA 4.0	0 m 19 s	0 m 3 s	0 m 4 s	189.55
ENV_100K.fna				
CUDASW++ 3.0	104 m 49 s	405 m 54 s	2 m 8 s	59.77
MIRA 4.0	2 m 49 s	0 m 36 s	0 m 35 s	2224.43

<sup>a</sup>The time or difference between a beginning time and an ending time of the program

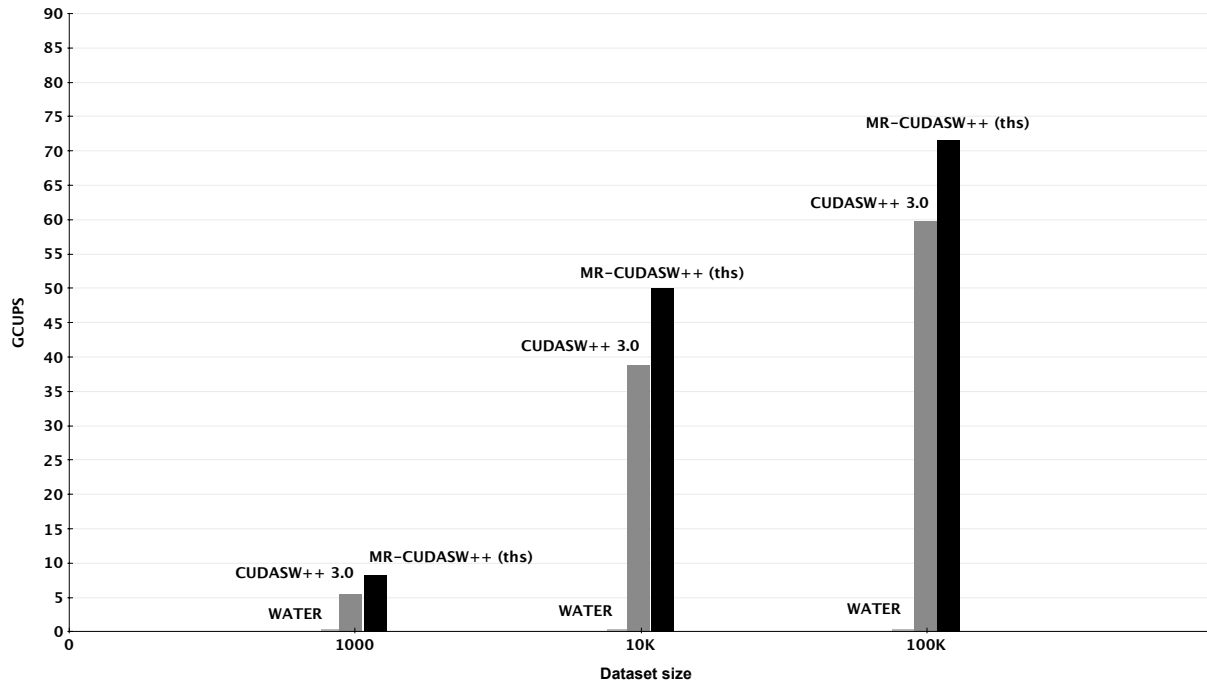
<sup>b</sup>Total amount of CPU-time that the process spent in user mode.

<sup>c</sup>Total amount of CPU-time that the process spent in kernel mode

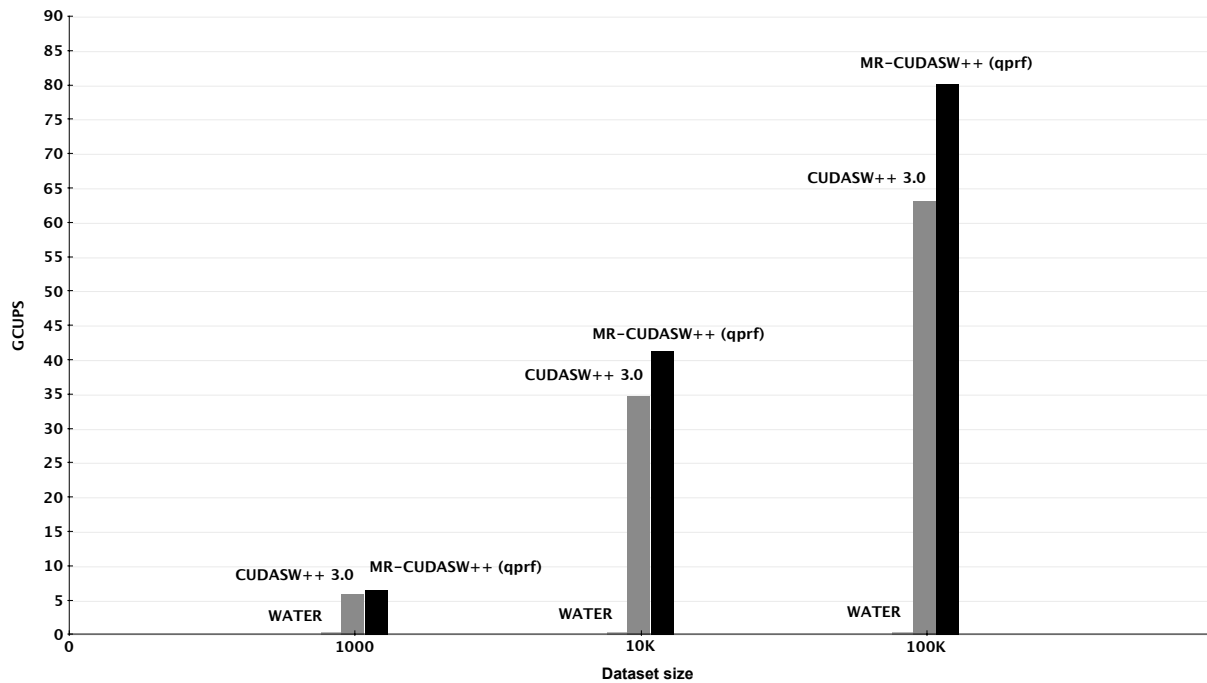
**Figure 6.5:** Effects of modifications on improving the performance of CUDASW++ 3.0 aligning simulated datasets of varying sizes. “ths” represents the modified software implementing the new thread scheduling and workload distribution techniques.



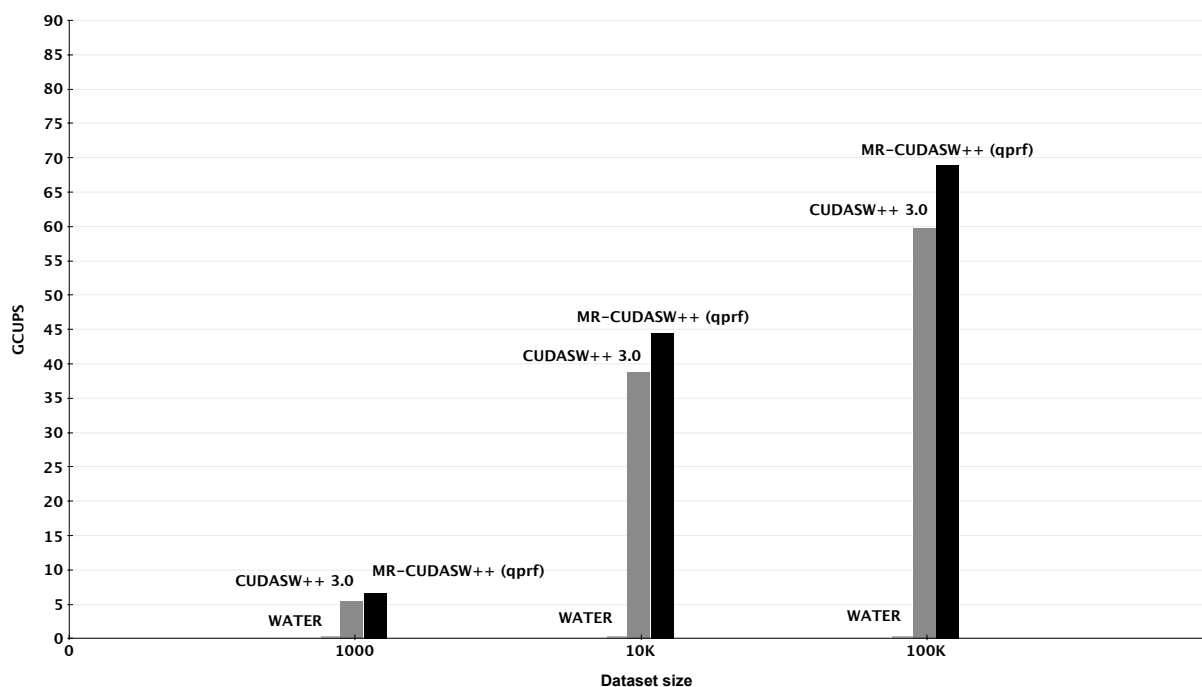
**Figure 6.6:** Effects of modifications on improving the performance of CUDASW++ 3.0 aligning real WGS datasets of varying sizes. “ths” represents the modified software implementing the new thread scheduling and workload distribution techniques.



**Figure 6.7:** Effects of modifications on improving the performance of CUDASW++ 3.0 aligning simulated datasets of varying sizes. “qprf” represents the modified software implementing striped query profile technique



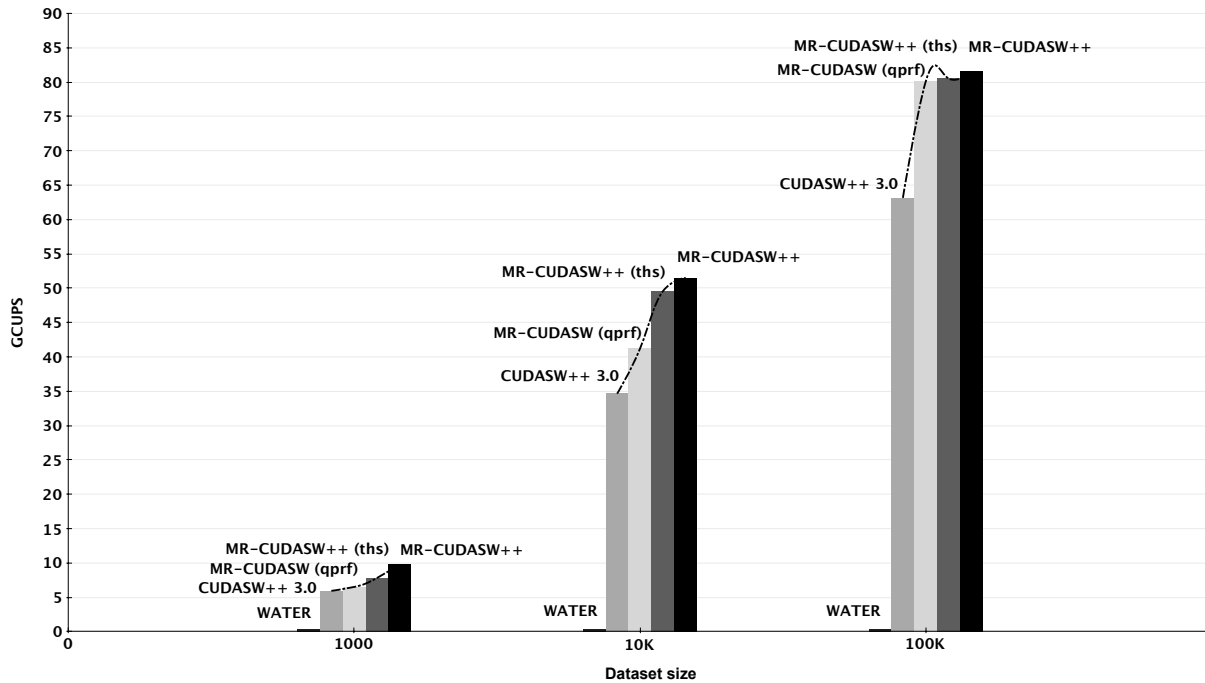
**Figure 6.8:** Effects of modifications on improving the performance of CUDASW++ 3.0 aligning real WGS datasets of varying sizes. “qprf” represents the modified software implementing striped query profile technique.



### 6.3 Evaluating MR-CUDASW

As discussed above, various modifications were made to improve the performance of CUDASW++ 3.0 (in order to replace this algorithm with the overlap determination technique used in MIRA assembly program). In this regard, workload distribution, query profile construction, and thread scheduling techniques implemented in CUDASW++ 3.0 were replaced by custom methods specifically designed to handle medium-length sequences. Effects of each of these modifications were measured in the previous sections. Eventually, all these modifications were combined together to create MR-CUDASW: the final improved version of CUDASW++ 3.0, specifically designed to handle medium-length sequences. Figures 6.9 and 6.10 illustrate the improvement in the performance of MR-CUDASW after each step. The performance of the final software is also presented.

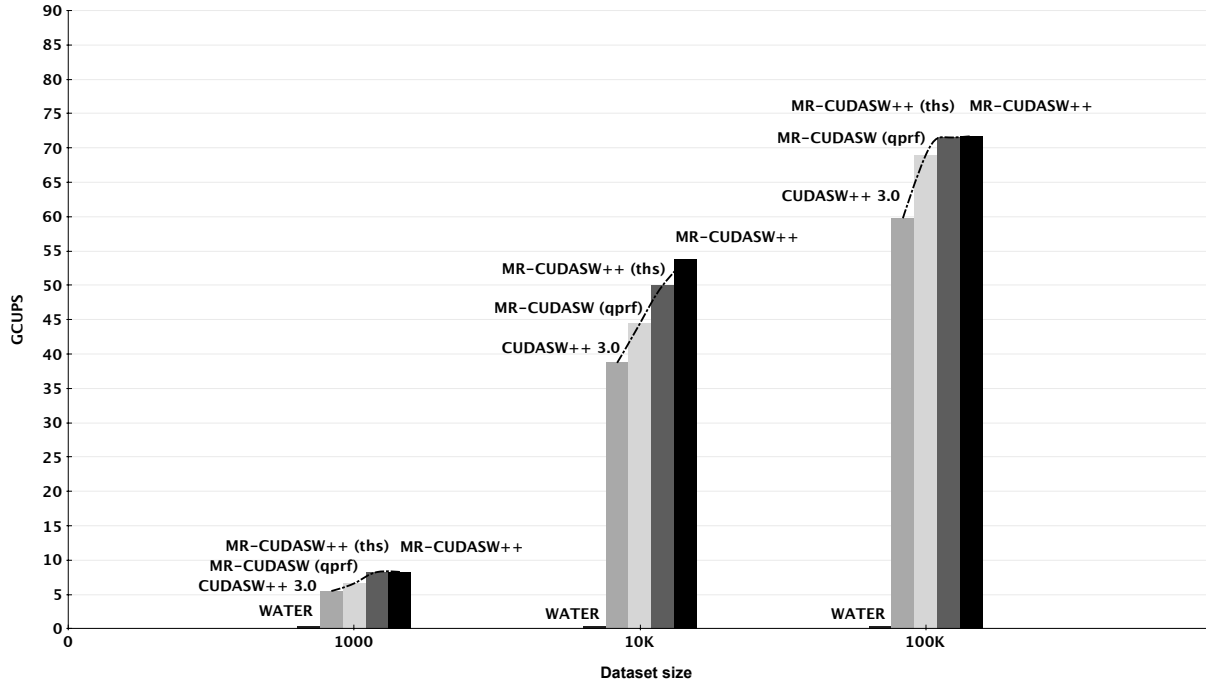
**Figure 6.9:** Effects of modifications on improving the performance of CUDASW++ 3.0 aligning simulated datasets of varying sizes. “qprf” represents the version of MR-CUDASW implementing the striped query profile technique. “ths” represents the version of MR-CUDASW with the modified thread scheduling and workload distribution techniques.



After confirming the correctness of the alignment scores produced by this software using the technique introduced in Section 5.2.3, the performance of the resultant software was compared with MIRA’s overlap detection phase. As mentioned before, however, due to lack of documentation and inadequate modularity in the software design, MIRA’s overlap determination stage could not be extracted from MIRA package. The performance of the whole MIRA’s assembly process, hence, was reported.

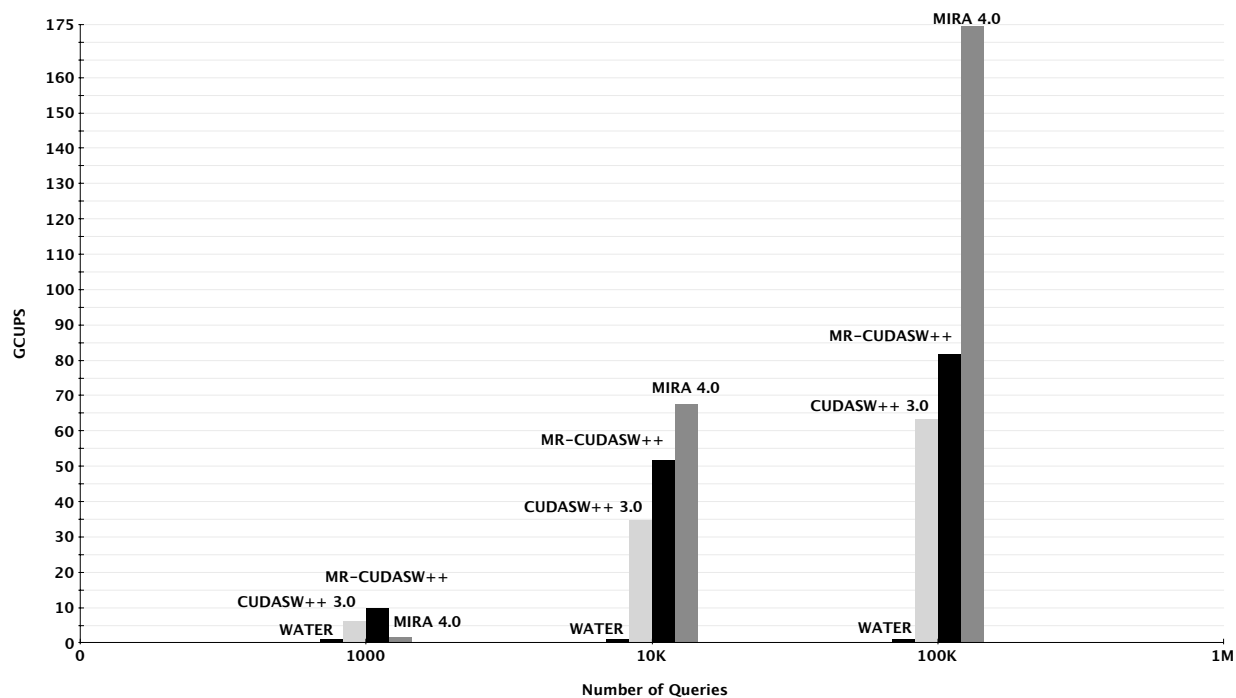
To this end, performances of all algorithms were compared by using simulated and real WGS datasets of varying sizes. Figures 6.11 and 6.12 illustrate the performance of all the evaluated algorithms for varying

**Figure 6.10:** Effects of modifications on improving the performance of CUDASW++ 3.0 aligning real WGS datasets of varying sizes. “qprf” represents the version of MR-CUDASW implementing the striped query profile technique. “ths” represents the version of MR-CUDASW with the modified thread scheduling and workload distribution techniques.

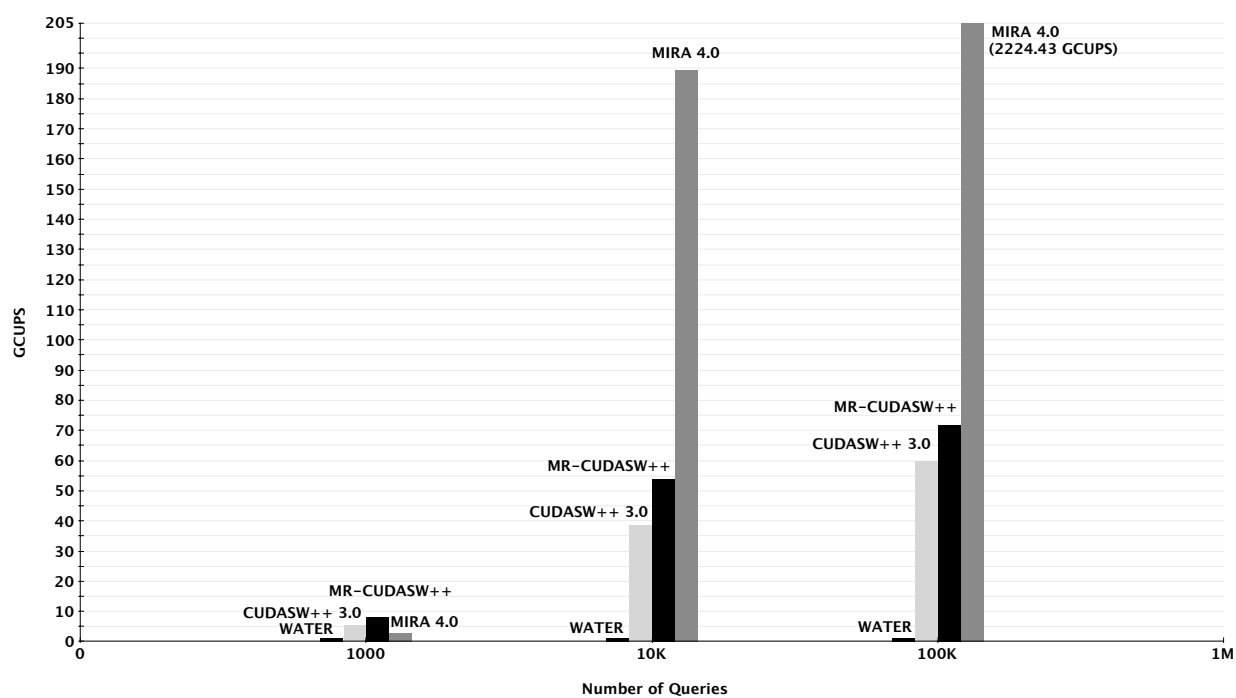


dataset sizes. On GTX680 graphic card (with the Kepler architecture), MR-CUDASW yields an average performance of 46.09 GCUPS, with a maximum of 81.53 GCUPS. EMBOSS water achieve nearly constant performance for all dataset sizes. CUDASW++ 3.0 has an average performance of 34.95 GCUPS on GTX680, while MIRA yields an average performance of 443.37 GCUPS, with a maximum of 2224.43 GCUPS using a 2.66 Quad core Intel Xeon CPU (256 KB L2 Cache, and 8 MB L3 Cache) and 16 GB RAM. MR-CUDASW is superior to both CUDASW++ 3.0 and EMBOSS water for every dataset size. MR-CUDASW++ on GTX 680 runs on average 1.40 times faster than CUDASW++ 3.0 and 65,854 times faster than EMBOSS water, while running 9.61 times slower than MIRA 4.0 on average. Full results of this comparison are presented in Appendix C.

**Figure 6.11:** Performances of GPU-accelerated alignment software tools and MIRA when determining overlaps between reads in simulated datasets of varying sizes on a GTX 680 Kepler architecture GPU.



**Figure 6.12:** Performances of GPU-accelerated alignment software tools and MIRA when determining overlaps between reads in real WGS datasets of varying sizes on a GTX 680 Kepler architecture GPU.



## CHAPTER 7

### CONCLUSION AND DISCUSSION

This research work aimed to determine the possibility of improving the performance of the overlap determination stage of an OLC-based assembler by using a GPU-based implementation of the Smith-Waterman algorithm, given the current state of GPU technology.

To this end, an existing assembly program, which met a number of criteria such that modification and improvement of the software is possible, was selected (i.e. MIRA) (Chapter 4). By taking advantage of parallelism we sought to improve the performance of the overlap determination stage of the chosen assembler by better utilizing computational resources (i.e. GPUs) available on a commodity desktop computer. For this purpose, a set of GPU-accelerated alignment programs that can be used to accurately find potential overlaps between each pair of sequences from a given list in a timely manner was determined. These algorithms were evaluated in order to pick the best software suitable for aligning medium-length nucleotide reads (Chapter 5). Techniques used in the chosen GPU-accelerated alignment software (i.e. CUDASW++ 3.0) were explored, adapted, and expanded in order to improve the performance of the software. As a result, various methods were implemented and the accuracy of the results were evaluated. The research led to the creation of MR-CUDASW, the final improved version of CUDASW++ 3.0, specifically designed to handle medium-length sequences. Finally, the performance of the resultant software, the sequential SW algorithm and the technique used by MIRA assembler were compared in order to assess the efficacy of the software and determine whether or not replacing MIRA's overlap determination technique would be advantageous. This chapter summarizes the results presented in Chapters 4 and 6. It provides discussion concerning selected aspects of MR-CUDASW and suggests possibilities for future work. Section 7.1 provides a more detailed review of MR-CUDASW, and Section 7.2 briefly highlights the major contributions of this thesis (these contributions are presented in *italic form*), gives some concluding remarks, and comments about the general applicability of this software. Finally, Section 7.3 discusses some of the future work that could be performed to further enhance the results.

#### 7.1 Conclusion and remarks

As presented in Chapter 2, the major objective of this thesis was to determine the possibility of improving the performance of the overlap determination stage of an OLC-based assembler (e.g. MIRA) by using a GPU-based implementation of the Smith-Waterman algorithm, given the current state of GPU technology.



We aimed to take advantage of the highly parallel many-core processor architecture of GPUs in order to incorporate the fastest, most sensitive method of identifying overlaps between two sequences into OLC-based assembly software that can be run on commonly available inexpensive hardware. This section briefly reviews the foundation, design, and implementation of this program. The conclusions and the major contribution of this research work are highlighted within the running synopsis.

This first step of this study was identification of existing assembly software that has the potential to be extended as a parallelized and optimized assembler for medium-length nucleotide reads sampled from highly complex environments with small sequence length deviation. To this end, a group of assemblers that are currently used for assembly of medium-length nucleotide data were evaluated. The applicability and performance of these assemblers were compared utilizing simulated data of different microbial community complexities (low, medium and high complexities, abbreviated as LC, MC and HC, respectively) as well as real DNA sequence fragments obtained from randomly selected whole-community DNA using Ion Torrent technology. Considering the computational time, maximum random access memory (RAM) occupancy, assembly accuracy and integrity, and the presence of a program's source and its maintainability and modularity, our study identified *MIRA as the best potential assembly software* that could meet our performance expectations while having grounds for improvement and modifications. For this purpose, several independent sequential modules implemented in MIRA (Figure 1.1) were considered to be replaced with scalable replacements. Specifically, the most compute-intensive portion of the application (i.e. the alignment step) was selected for replacement by an alternative GPU-based implementation of the Smith-Waterman algorithm.

In this regard, a set of GPU-accelerated alignment software that could be used to accurately find potential overlaps between each pair of sequences from a given list in a timely manner was identified. The suitability of these software tools was determined based on our review of their published descriptions and evaluation of their performances. To this end, three simulated nucleotide datasets as well as three real nucleotide datasets containing 1000, 10,000, and 100,000 reads were used to benchmark these algorithms. Since most of these algorithms were originally designed to take advantage of many-core parallelism on GPUs with Fermi architecture and could not be readily transformed onto a machine with Kepler architecture GPU, our evaluation was carried out in two steps. First all the software tools designed based on Fermi architecture were compared against each other. The fastest sequence aligner running on a Fermi architecture GPU was then modified to run on a Kepler architecture and benchmarked against CUDASW++ 3.0. Conclusively, *CUDASW++ 3.0 showed the best performance among the other benchmarked GPU-based implementations of the Smith-Waterman algorithm* (Figures 6.3 & 6.4).

In the next step, to answer the question of whether or not it is possible to improve the performance of an OLC-based assembly software by using GPU-based acceleration techniques, the performance of MIRA's overlap determination stage was compared with the fastest GPU-based implementation of the Smith-Waterman algorithm (Tables 6.2 & 6.3). As this evaluation illustrated, *the average elapsed real time of MIRA whole assembly process was significantly smaller than the average elapsed real time of CUDASW++ 3.0*. Hence the

performance of CUDASW++ 3.0 still needed to be improved in order to replace MIRA’s overlap determination technique.

For this purpose, CUDASW++ 3.0 was closely analyzed and the technical features that made this software far faster than the other benchmarked software were identified. Some of these techniques were further extended to improve the performance of this software tool and some were replaced. Over the course of this phase, workload distribution, query profile construction, and thread scheduling techniques implemented in CUDASW++ 3.0 were replaced by techniques specifically customized to handle medium-length reads.

This research’s focus on a distinct range of sequence lengths and its use of the expanded per-thread local memory size of the Kepler architecture eliminated the necessity of implementing the intra-task parallelization technique. In addition, instead of distributing the workload only according to the compute power of CPUs and GPUs (the technique employed in CUDASW++ 3.0), the workload distribution was modified in a way that distributes the workload based on the length of the query sequence, the length of the subject sequence, and the total sequence length deviation calculated for the dataset, as well as the compute power of the CPUs and GPUs. Using this method of thread scheduling and workload distribution, the computation waste of CPU and GPU SIMD instructions was avoided as all alignments in all lanes are completed at the same time. This technique also prevented the computational imbalance between threads within a warp and a thread block. Conclusively, *this new workload distribution technique yielded an average performance of 44.62 GCUPS, 1.28 times faster compare to CUDASW++ 3.0* (see Figures 6.9 & 6.10).

Although modification of the methods for workload distribution and thread scheduling led to significant speed up of the software tool, the performance of MIRA whole assembly process was still far better than the performance of the improved version of CUDASW++ 3.0. More improvements were then required. It was known that CUDASW++ 3.0 made use of the Rognes and Seebergs technique to construct query profiles, parallel to the query sequence for each possible residue (Figure 5.2 part B). Using this technique, four substitution scores were realized using only one texture fetch, thus significantly improving texture memory throughput. It has been noted [53] that the execution time of this method is heavily dependent on the length of the query string and the database, the scoring matrix and gap penalties. The distinct range of target sequences in this study provided an opportunity to explore the effects of the striped implementation of Smith Waterman on a Kepler architecture GPU, without being concern about the this technique’s high dependency on query length. This implementation of the Smith-Waterman algorithm was introduced by Farrar et al. [50] and, although highly dependent on query length, is believed to be the fastest implementation of this algorithm. As a result of improving and customizing the already-implemented striped query profile construction method for Kepler architecture and taking advantage of this architecture’s expanded texture memory, *the improved version illustrated an average performance of 40.66 GCUPS, gaining 1.17 times speedup compared to the original version of the software*. Figures 6.9 and 6.10 show the effects of this modification.

Eventually, all these modifications were combined together to create MR-CUDASW, the final improved version of CUDASW++ 3.0, specifically designed to handle medium-length sequences. To ensure the cor-

rectness of the resultant tool, it was evaluated against the EMBOSS sequential implementation of the Smith-Waterman algorithm. All the resultant scores from two programs were compared using a custom shell script and *in all the cases scores were identical* (data not shown). In the next step, the performance of the resultant software was compared with MIRA’s overlap detection phase. As mentioned before, however, due to lack of documentation and inadequate modularity in the software design, MIRA’s overlap determination stage could not be extracted from MIRA package. The performance of MIRA’s whole assembly process, hence, was reported. To this end, performances of all algorithms were compared by using simulated and real WGS datasets of varying sizes. Conclusively, MR-CUDASW yields an average performance of 46.09 GCUPS, with a maximum of 81.53 GCUPS. EMBOSS water achieve nearly constant performance for all database sizes ( $< 0.01$  GCUPS). CUDASW++ 3.0 has an average performance of 34.95 GCUPS, while MIRA yields an average performance of 443.37 GCUPS, with a maximum of 2224.43 GCUPS. *MR-CUDASW is superior to both CUDASW++ 3.0 and EMBOSS Water for every dataset size. MR-CUDASW++ runs on average 1.40 times faster than CUDASW++ 3.0 and 65,854 times faster than EMBOSS Water, while running 9.61 times slower than MIRA 4.0 on average.* Figures 6.11 and 6.12 illustrate the performance of all evaluated algorithms for varying dataset sizes.

Our techniques exploit the GPU cores efficiently by distributing the sequences with large deviation to the CPU. On average, compared to the aforementioned performance of CUDASW++ 3.0, *the alternative distribution technique can improve the performance by 17% for medium-length queries. In general, for medium-length queries, more performance gains can be realized from the stripped query profile because it can reduce the number of texture fetches by half and use fewer bitwise operations per cell.* Considering that MR-CUDASW is targeting a specific range of sequence lengths and *the new workload distribution technique in which sequences with large sequence length deviation are assigned to CPUs, the length dependency of the stripped query profile construction method can be dismissed and 28% performance gain can be achieved as the result of replacing the sequential query profile construction technique with the stripped method.*

## 7.2 Discussion

MR-CUDASW is the first highly parallel solution that has been specifically optimized to process medium-length nucleotide reads (DNA/RNA) from modern sequencing machines (i.e. Ion Torrent). Results show that the software reaches up to 82 GCUPS (Giga Cell Updates Per Second) on a single GPU running on a commodity desktop hardware and as a result it is the fastest tool in its class. Despite being designed for performing the Smith-Waterman algorithm on medium-length nucleotide sequences, MR-CUDASW also presents great potential for heterogeneous computing with CUDA-enabled GPUs and is expected to make contributions to any other research problems that require sensitive pairwise alignment to be applied to a large number of reads.

Our results utilizing GPUs show that it is possible to improve the performance of biological algorithms

by making full use of the compute characteristics of the underlying commodity hardware and further, our results are especially encouraging since GPU performance is growing faster than multi-core CPUs [75].

### 7.3 Future work

This section describes further work that could be carried out to improve on or extend this work.

- Although specifically designed to handle nucleotide sequences, none of the techniques implemented in this work limits MR-CUDASW to nucleotide sequences. This program can be readily modified to perform Smith Waterman algorithm on amino acids sequences.
- In addition to the performance evaluation of hybrid CPU-GPU parallelism, the evaluation of relative performance of CPU computation to GPU computation can be used to further measure the balance of the implemented workload distribution technique. To this end, the runtime of CPU and GPU computation should be recorded in order to calculate their performance relative to a specific workload.
- All the performance evaluations presented in this thesis were carried out on a single-GPU GTX680 card. Since MR-CUDASW is considered to be the improved version of CUSAW++ 3.0, specifically adapted to accommodate medium-length nucleotide sequences, it, potentially, has the ability to run on dual-GPU Geforce graphic cards. The lack of such hardware prevented us from further investigations. Better results might be gathered in a performance comparison between MR-CUDASW and MIRA on machine equipped with dual-GPU graphic cards.
- As discussed by Liu [75], the optimal alignment scores of the SW algorithm are biased by sequence length and composition. The Z-value has been proposed to estimate the statistical significance of these scores. However, the computation of Z-value requires performing a large set of pairwise alignments between random permutations of the sequences compared, which is highly time-consuming. The acceleration of a Z-value computation with on CUDA-enabled graphic card can therefore be a part of future work. For more information about Z-value, please refer to the work by Comet et al. [79], Bastien et al. [80], and Peris et al. [81].

## REFERENCES

- [1] Fredrick Sanger, Smith Nicklen, and Arnold Coulson. DNA sequencing with chain-terminating inhibitors. *Proceedings of the National Academy of Sciences*, 74(12):5463–5467, December 1977.
- [2] Craig J. Venter, Karin Remington, John F. Heidelberg, Aaron L. Halpern, Doug Rusch, Jonathan A. Eisen, Dongying Wu, Ian Paulsen, Karen E. Nelson, William Nelson, Derrick E. Fouts, Samuel Levy, Anthony H. Knap, Michael W. Lomas, Ken Neelson, Owen White, Jeremy Peterson, Jeff Hoffman, Rachel Parsons, Holly Baden-Tillson, Cynthia Pfannkoch, Yu-Hui Rogers, Hamilton O. Smith. Environmental genome shotgun sequencing of the Sargasso Sea. *Science*, 304(5667):66–74, April 2004.
- [3] Daniel N. Frank. Growth and development symposium: Promoting healthier humans through healthier livestock. *Journal of Animal Science*, 89(3):835–844, 2011.
- [4] The NIH HMP Working Group. The NIH human microbiome project. *Genome Research*, 19(12):2317–2323, December 2009.
- [5] Norman Grossblatt. *The new science of metagenomics: revealing the secrets of our microbial plan*. The National Academies press, 2007.
- [6] Falk Warnecke and Matthias Hess. A perspective: Metatranscriptomics as a tool for the discovery of novel biocatalysts. *Journal of Biotechnology*, 142(1):91–95, June 2009.
- [7] Rudolf Amann, Wolfgang Ludwig, and Karl H. Schleifer. Phylogenetic identification and in situ detection of individual microbial cells without cultivation. *Microbiological Reviews*, 59(1):143–169, March 1995.
- [8] Johannes Dröge and Alice C. McHardy. Taxonomic binning of metagenome samples generated by next-generation sequencing technologies. *Briefings in Bioinformatics*, 13(6):646–655, 2012.
- [9] John C. Wooley, Adam Godzik, and Iddo Friedberg. A primer on metagenomics. *Journal of Computational Biology*, 6(2):e1000667, February 2010.
- [10] Simon Dear, Richard Durbin, LaDeana Hillier, Gabor Marth, Jean Thierry-Mieg, and Richard Mott. Sequence assembly with CAFTOOLS. *Genome Research*, 8:260/267, 1998.
- [11] Chris Armen and Clifford Stein. Short superstrings and the structure of overlapping strings. *Journal of Computational Biology*, 2(2):307–332, 1995.
- [12] Bastien Chevreux. *MIRA: An automated genome and EST assembler*. PhD thesis, German Cancer Research Center Heidelberg. June 2005.
- [13] David Hernandez, Patrice François, Laurent Farinelli, Magne Østerås, and Jacques Schrenzel. De novo bacterial genome sequencing: Millions of very short reads assembled on a desktop computer. *Genome Research*, 18(5):802–809, May 2008.
- [14] Pavel A. Pevzner, Haixu Tang, and Michael Waterman. An Eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2001.
- [15] Daniel R. Zerbino. *Genome assembly and comparison using de Bruijn graphs*. PhD thesis, Darwin College, University of Cambridge. September 2009.
- [16] Adam Phillippy, Micheal Schatz, and Mihai Pop. Genome assembly forensics: finding the elusive mis-assembly. *Genome Biology*, 9(3):55–67, March 2008.

- [17] Wenyu Zhang, Jiajia Chen, Yang Yang, Yifei Tang, Jing Shang and Bairong Shen. A practical comparison of de novo genome assembly software tools for next-generation sequencing technologies. *PLoS ONE*, 6(3):e17915, March 2011.
- [18] Liu Weiguo, Bertil Schmidt, Gerrit Voss, and Wolfgang Muller-Wittig. Streaming algorithms for biological sequence alignment on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 18(9):1270–1281, 2007.
- [19] Sébastien Boisvert, Frédéric Raymond, Élénie Godzaridis, François Laviolette, and Jacques Corbeil. Ray Meta: scalable de novo metagenome assembly and profiling. *Genome Biology*, 13(12):122–137, December 2012.
- [20] Marketa Zvelebil and Jeremy Baum. *Understanding Bioinformatics*. Garland Science Press, Taylor and Francis Group, 2008.
- [21] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley, 2010.
- [22] Niranjana Nagarajan and Mihai Pop. Sequence assembly demystified. *Nature Reviews Genetics*, 14:157–167, March 2013.
- [23] Saul Needleman and Christian Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.
- [24] Temple F. Smith and Michael S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, March 1981.
- [25] John W. Wilbur and David J. Lipman. Rapid similarity searches of nucleic acid and protein data banks. *Proceedings of the National Academy of Sciences*. 80(3):726–730, February 1983.
- [26] Stephen F Altschul, Warren Gish, Webb Miller, Eugene Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, October 1990.
- [27] Burkhard Rost, Chris Sander, and Reinhard Schneider. PHD – an automatic mail server for protein secondary structure prediction. *Computer Applications in the Biosciences*, 10(1):53–60, February 1994.
- [28] Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme Mitchison. *Biological sequence analysis*. Cambridge University press, 1998.
- [29] Margaret O. Dayhoff, and Robert M. Schwartz. Chapter 22: A model of evolutionary change in proteins, In *Atlas of protein sequence and structure*. National Biomedical Research Foundation, 1978.
- [30] AAindex. Amino acid index database. July 2014.
- [31] Arthur M. Lesk. *Introduction to bioinformatics*. Oxford University press, 2006.
- [32] Peter H. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *SIAM Journal on Applied Mathematics*, 26(4):787–793, 1974.
- [33] Osamu Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162:705–708, 1982.
- [34] Jonathan Pevsner. *Bioinformatics and functional genomics*. John Wiley and Sons, 2009.
- [35] Stephen F. Altschul. *Global and local sequence alignment, Lecture slides*. Department of Computer Science, University of Maryland. October 2011.
- [36] Lusheng Wang and Tao Jiang. On the complexity of multiple sequence alignment. *Journal of Computational Biology*, 1(4):337–348, 1994.
- [37] Victor Simossis, Jens Kleinjung, and Jaap Heringa. Chapter 3: An overview of multiple sequence alignment. In *Current Protocols in Bioinformatics*, 2003.

- [38] David J. Lipman and William R. Pearson. Rapid and sensitive protein similarity searches. *Science*, 227(4693):1435–1441, March 1985.
- [39] Makoto Hirosawa, Yasushi Totoki, Masaki Hoshida, and Masato Ishikawa. Comprehensive study on iterative algorithms of multiple sequence alignment. *Computer Applications in the Biosciences*, 11(1):13–18, February 1995.
- [40] Blaise Barney. *Introduction to parallel computing, Tutorial series*. Lawrence Livermore National Laboratory. July 2013.
- [41] Nicholas Carriero and David Gelernter. *How to write parallel programs*. Massachusetts Institute of Technology, 1999.
- [42] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972.
- [43] John Von Neumann. *First draft of a report on the EDVAC*. Moore School of Electrical Engineering, University of Pennsylvania 1945.
- [44] Michael Catherwood. Modified harvard architecture processor having data memory space mapped to program memory space with erroneous execution protection. US Patent Application. 09/870,460. January 2003.
- [45] David Luebke, Mark Harris, Jens Krüger, Tim Purcell, Naga Govindaraju, Ian Buck, Cliff Woolley, and Aaron Lefohn. GPGPU: general purpose computation on graphics hardware. In *ACM SIGGRAPH 2004 course notes*, pages 33+, New York, NY, USA, 2004. ACM.
- [46] John Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Kruger, Aaron Lefohn, and Timothy J. Purcell. A Survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, March 2007.
- [47] Victor Lee, Changkyu Kim, Jatin Chhugani, Micheal Deisher, Daehyun Kim, Anthony Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *Proceedings of the 37<sup>th</sup> annual international symposium on computer architecture*, Volume 38 of *ISCA 2010*, pages 451–460, New York, NY, USA, June 2010. ACM.
- [48] Timothy F. Oliver, Bertil Schmidt, and Douglas L. Maskell. Reconfigurable architectures for bio-sequence database scanning on FPGAs. *Circuits and systems II: Express briefs, IEEE Transactions*, 52(12):851–855, 2005.
- [49] Tim Oliver, Bertil Schmidt, Darran Nathan, Ralf Clemens, and Douglas Maskell. Using reconfigurable hardware to accelerate multiple sequence alignment with ClustalW. *Bioinformatics*, 21(16):3431–3432, 2005.
- [50] Michael Farrar. Striped Smith-Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics*, 23(2):156–161, January 2007.
- [51] Svetlin A. Manavski and Giorgio Valle. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics*, 9(Suppl 2):S10–9, March 2008.
- [52] Adam Wozniak. Using video-oriented instructions to speed up sequence comparison. *Computer Applications in the Biosciences*, 13(2):145–150, April 1997.
- [53] Torbjørn Rognes, Erling Seeberg. Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, 16(8):699–706, August 2000.
- [54] Adam M. Szalkowski, Christian Ledergerber, Philipp Krähenbühl, and Christophe Dessimoz. SWPS3 – Fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and x86/SSE2. *BMC Research Notes*, 1(1):107+, 2008.

- [55] *CUDA C programming guide*. NVIDIA official technical report. July 2013.
- [56] Jonh Nickolls, Ian Buck, Micheal Garland, and Kevin Skadron. Scalable parallel programming with CUDA. In *ACM SIGGRAPH 2008 classes*, pages 1–14, New York, NY, USA, 2008. ACM.
- [57] Yongchao Liu, Adrianto Wirawan, and Bertil Schmidt. CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. *BMC Research Notes*, 2(1):73+, 2009.
- [58] James Wang. From fermi to kepler. *Nvision Magazine*, March 2012.
- [59] Yu Peng, Henry Leung, Siu-Ming Yiu, and Francis Chin. IDBA-UD: a de novo assembler for single-cell and metagenomic sequencing data with highly uneven depth. *Bioinformatics*, 28(11):1420–1428, June 2012.
- [60] Toshiaki Namiki, Tsuyoshi Hachiya, Hideaki Tanaka, Yasubumi Sakakibara. MetaVelvet: an extension of Velvet assembler to de novo metagenome assembly from short sequence reads. *Nucleic Acids Research*, 40(20):e155, November 2012.
- [61] R. L. Warren, G Sutton, S Jones, and R Holt. Assembling millions of short DNA sequences using SSAKE. *Bioinformatics*, 23(4):500–501, February 2007.
- [62] Miguel Pignatelli and Andres A. Moya. Evaluating the fidelity of de novo short read metagenomic assembly using simulated data. *PLoS ONE*, 6(5):e19984+, May 2011.
- [63] Kim Pruitt, Tatiana Tatusova, and Donna R. Maglott. NCBI reference sequence (RefSeq): a curated non-redundant sequence database of genomes, transcripts and proteins. *Nucleic Acids Research*, 1(33), 2005.
- [64] Stephen E. Johnson, Brett Trost, Jeffrey R. Long, and Anthony Kusalik. A better sequence-read generator program for metagenomics. *European Molecular Biology Network Journal*, 19(A):49–50, 2014.
- [65] Hannon Lab. *FASTX Toolkit*. Official Manual.
- [66] Nicholas J. Loman, Raju V. Misra, Timothy J. Dallman, Chrystala Constantinidou, Saheer E. Gharbia, John Wain, and Mark J. Pallen. Performance comparison of benchtop high-throughput sequencing platforms. *Nature Biotechnology*, 30(5):434–439, April 2012.
- [67] Ben Langmead and Steven L. Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature Methods*, 9(4):357–9, April 2012.
- [68] I Li, Warren Shum, and Kevin Truong. 160-fold acceleration of the Smith-Waterman algorithm using a field programmable gate array (FPGA). *BMC Bioinformatics*, 8(1):185+, June 2007.
- [69] Michael S. Farrar. Optimizing Smith-Waterman for the cell broadband engine. <http://cudasw.sourceforge.net/sw-cellbe.pdf> July 2014
- [70] Kwong Adrianto, Keong K. Chee, Nim Hieu, and Bertil Schmidt. CBESW: sequence alignment on the Playstation 3. *BMC Bioinformatics*, 9(1):377+, September 2008.
- [71] Yongchao Liu, Adrianto Wirawan, and Bertil Schmidt. CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. *BMC Bioinformatics*, 14(1):117+, April 2013.
- [72] Wojciech Frohberg, Michal Kierzyńska, Jacek Blazewicz, Peter Gawron, and Peter Wojciechowski. G-DNA – a highly efficient multi-GPU/MPI tool for aligning nucleotide reads. *Bulletin of the Polish Academy of Sciences*, 61(4):989–992, 2013.
- [73] Alan Bleasby. *Smith-Waterman local alignment of sequences*. European Bioinformatics Institute, Cambridge, UK. 3rd edition, October 2000.



- [74] Torbjørn Rognes. Faster Smith-Waterman database searches with inter-sequence SIMD parallelisation. *BMC Bioinformatics*, 12(1):221+, June 2011.
- [75] Yongchao Liu, Adrianto Wirawan, and Bertil Schmidt. CUDASW++2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions. *BMC Research Notes*, 3(1):93+, 2010.
- [76] Torbjørn Rognes, Erling Seeberg. Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, 16(8):699–706, August 2000.
- [77] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro Magazine*, 28(2):39–55, March 2008.
- [78] Jeremiah van Oosten. *Introduction to CUDA 5.0*. NVIDIA Official Release Note. NVIDIA, October 2012.
- [79] Jean P. Cometa, Jean C. Audea, Emilie Glémata, Jill L. Rislerb, Antonie Hénautb, Piotr P. Slonimskib, and Ju Codani. Significance of Z-value statistics of Smith-Waterman scores for protein alignments. *Computers & Chemistry*, Elsevier, 23(3-4):317–334, 1999.
- [80] Olivier Bastien, Jean-Christophe Aude, Sylvaine Roy, and Eric Marchal. Fundamentals of massive automatic pairwise alignments of protein sequences: theoretical significance of Z-value statistics. *Bioinformatics*, 20(4):534–537, 2004.
- [81] Guillermo Peris, and Andres Marzal. A screening method for Z-Value assessment based on the normalized edit distance. *Lecture Notes in Computer Science*, 5518:1154–1161, 2009.
- [82] *PTX: Parallel Thread Execution*, NVIDIA Corporation, 1.0 edition, 2007.

# APPENDIX A

## COMPARISON OF ASSEMBLY SOFTWARE

### A.1 Evaluating various assembly software using low complexity simulated dataset

**Table A.1:** Size statistics of the results of using various assembly program on low complexity simulated data.

	IDBA-UD	Newbler	SSAKE	MIRA	RayMéta	MetaVelvet
Assembled total size (Bp)	31,241,384	23,468,395	24,528,006	31,494,293	28,263,944	28,762,606
Number of total contigs	29,040	24,387	33,376	32,685	64,499	43,296
Number of major contigs <sup>a</sup>	2638	1357	1266	2406	1634	1703
Size of the longest contig (Bp)	245,266	192,000	165,859	214,691	72,596	149,340
Average contig Size (Bp)	1075	962	734	963	438	664
N50 size (Bp)	2045	3458	2684	1935	644	1281
N80 size (Bp)	520	453	318	479	197	327
Average number of reads per contig	6.76	6.05	4.62	6.06	2.75	4.17
Number of short contigs <sup>b</sup>	0	1401	53	92	64,214	4069

<sup>a</sup>Number of contigs longer than 10\*(average read size)

<sup>b</sup>Number of short contigs shows the number of contigs with a length smaller than the average read size. These contigs are filtered out from our basic evaluation, and are not included in the "total number of contigs".

**Table A.2:** Percentage of contigs produced by various assembly software, assembling low complexity simulated data, mapping to the reference genomes.

% of contigs having	IDBA-UD	Newbler	SSAKE	MIRA	RayMéta	MetaVelvet
≥ 60% identity	98.11%	99.98%	99.97%	99.75%	99.95%	99.96%
≥ 80% identity	94.55%	99.80%	99.94%	98.76%	99.95%	99.85%
≥ 90% identity	92.81%	99.33%	99.59%	97.66%	99.81%	99.51%
≥ 95% identity	87.70%	95.07%	94.59%	92.91%	96.3%	93.24%
≥ 98% identity	81.55%	88.39%	90.22%	87.49%	93.71%	86.24%

## A.2 Evaluating various assembly software using medium complexity simulated dataset

**Table A.3:** Size statistics of the results of using various assembly programs on medium complexity simulated data.

	IDBA-UD	Newbler	SSAKE	MIRA	RayMéta	MetaVelvet
Assembled total size (Bp)	45,323,281	31,099,506	32,503,362	45,966,279	35,997,207	38,721,566
Number of total contigs	53,317	38,573	48,783	61,575	95,808	69,968
Number of major contigs <sup>a</sup>	3218	2083	1945	3681	2058	2295
Size of the longest contig (Bp)	190,588	191,991	165,499	204,271	67,122	149,340
Average contig Size (Bp)	850	806	666	746	375	553
N50 size (Bp)	1128	1843	1397	1041	422	858
N80 size (Bp)	430	380	297	399	178	271
Average number of reads per contig	5.34	5.07	4.19	4.69	2.36	3.48
Number of short contigs <sup>b</sup>	0	2548	90	223	126,147	7949

<sup>a</sup>Number of contigs longer than 10\*(average read size).

<sup>b</sup>Number of short contigs shows the number of contigs with a length smaller than the average read size. These contigs are filtered out from our basic evaluation, and are not included in the “total number of contigs”.

**Table A.4:** Percentage of contigs produced by various assembly software, assembling medium complexity simulated data, mapping to the reference genomes.

% of contigs having	IDBA-UD	Newbler	SSAKE	MIRA	RayMéta	MetaVelvet
≥ 60% identity	96.14%	99.95%	99.98%	99.72%	99.96%	99.95%
≥ 80% identity	90.55%	99.70%	99.91%	98.61%	99.95%	99.86%
≥ 90% identity	87.91%	99.10%	99.46%	97.34%	9.83%	99.43%
≥ 95% identity	81.61%	93.20%	93.47%	92.46%	96.32%	92.42%
≥ 98% identity	73.67%	83.89%	87.39%	85.38%	93.52%	84.46%

### A.3 Evaluating various assembly software using high complexity simulated dataset

**Table A.5:** Size statistics of the results of using various assembly program high complexity simulated data.

	IDBA-UD	Newbler	SSAKE	MIRA	RayMéta	MetaVelvet
Assembled total size (Bp)	86,986,932	58,408,513	52,715,872	91,903,971	75,025,609	153,414,832
Number of total contigs	123,875	109,008	118,855	142,052	287,690	553,507
Number of major contigs <sup>a</sup>	7648	4338	3285	8767	2044	4278
Size of the longest contig (Bp)	44,499	33,881	30,878	84,297	22,072	35,058
Average contig Size (Bp)	702	535	443	646	260	277
N50 size (Bp)	792	611	456	804	233	266
N80 size (Bp)	409	321	262	379	174	176
Average number of reads per contig	4.41	3.36	2.78	4.06	1.64	1.74
Number of short contigs <sup>b</sup>	0	7325	261	610	510,310	54,534

<sup>a</sup>Number of contigs longer than 10\*(average read size).

<sup>b</sup>Number of short contigs shows the number of contigs with a length smaller than the average read size. These contigs are filtered out from our basic evaluation, and are not included in the “total number of contigs”.

**Table A.6:** Percentage of contigs produced by various assembly software, assembling high complexity simulated data, mapping to the reference genomes.

% of contigs having	IDBA-UD	Newbler	SSAKE	MIRA	RayMéta	MetaVelvet
≥ 60% identity	97.67%	99.94%	99.97%	99.62%	99.98%	99.96%
≥ 80% identity	94.64%	99.62%	99.92%	98.23%	99.97%	99.92%
≥ 90% identity	90.80%	98.99%	99.52%	96.75%	99.87%	99.71%
≥ 95% identity	81.10%	93.26%	95.10%	91.51%	96.83%	94.69%
≥ 98% identity	69.15%	82.37%	90.18%	82.57%	93.39%	88.39%

## A.4 Evaluating various assembly software using real WGS dataset

**Table A.7:** Size statistics of the results of using various assembly program on real WGS data.

	IDBA-UD	Newbler	SSAKE	MIRA	RayMéta	MetaVelvet
Assembled total size (Bp)	44,419,747	21,436,696	166,712	40,699,034	233,027,595	67,618,696
Number of total contigs	109,800	54,555	538	85,226	1,102,935	273,128
Number of major contigs <sup>a</sup>	41	121	0	671	1	1
Size of the longest contig (Bp)	3289	6988	655	9374	2344	1961
Average contig Size (Bp)	404	392	309	477	211	247
N50 size (Bp)	385	399	308	505	215	227
N80 size (Bp)	325	287	271	334	201	206
Average number of reads per contig	2.09	2.03	1.60	2.47	1.09	1.28
Number of short contigs <sup>b</sup>	0	13,420	13	3683	2,738,596	245,926

<sup>a</sup>Number of contigs longer than 10\*(average read size).

<sup>b</sup>Number of short contigs shows the number of contigs with a length smaller than the average read size. These contigs are filtered out from our basic evaluation, and are not included in the "Number of total contigs".

# APPENDIX B

## CORE PTX SIMD ASSEMBLIES

To gain more speedup, CUDASW++ 3.0 implements the recurrence in Equation B.1 with PTX SIMD assembly instructions<sup>1</sup>. The code consists of ten assembly instructions for the recurrence and one instruction for obtaining the optimal local alignment score. For additions and subtractions, saturation instructions have been used to clamp the values to their appropriate signed ranges [71]. Listings B.1 shows the PTX SIMD assembly instructions as implemented by CUDASW++ 3.0.

Given two sequences  $S_a$  and  $S_b$  of lengths  $l_a$  and  $l_b$  respectively, the SW algorithm computes the similarity score  $H(i, j)$  of two sequences ending at position  $i$  and  $j$  of  $S_a$  and  $S_b$ , respectively. This figure show the computation of  $H(i, j)$  where,  $sbt$  is the character substitution cost table,  $\rho$  is the gap opening penalty and  $\sigma$  is the gap extension penalty.

$$\begin{aligned} E(i, j) &= \max \{E(i, j-1) - \sigma, H(i, j-1) - \rho - \sigma\} \\ F(i, j) &= \max \{E(i-1, j) - \sigma, H(i-1, j) - \rho - \sigma\} \\ H(i, j) &= \max \{0, E(i, j), F(i, j), H(i-1, j-1) + sbt(S_a[i], S_b[j])\} \end{aligned} \quad (B.1)$$

As it appears in the listing, every instruction operates on quads of 8-bit signed values, corresponding to four independent alignments. Variables  $h, n$  and  $h_e$  represent the alignment score vectors corresponding to matrix  $H$ , where  $h$  denotes the score vector of the four current cells,  $n$  the score vector of the four diagonal neighbours and  $h_e$  the score vector of the four upper neighbours. Variables  $e$  and  $f$  represent the score vectors corresponding to the matrices  $E$  and  $F$  respectively, and  $S$  stores.

**Listing B.1:** PTX SIMD assembly instructions.

---

```
#define ONE_CELL_COMP_QUAD(f, oe, ie, h, he, hd, sub, gapoe, gape, maxHH) \
asm("vsub4.s32.s32.s32.sat %0, %1, %2, %3;" : "=r"(f) : "r"(f), "r"(gape), "r"(0)); \
asm("vsub4.s32.s32.s32.sat %0, %1, %2, %3;" : "=r"(oe) : "r"(ie), "r"(gape), "r"(0)); \
asm("vsub4.s32.s32.s32.sat %0, %1, %2, %3;" : "=r"(h) : "r"(h), "r"(gapoe), "r"(0)); \
asm("vmax4.s32.s32.s32 %0, %1, %2, %3;" : "=r"(f) : "r"(f), "r"(h), "r"(0)); \
asm("vsub4.s32.s32.s32.sat %0, %1, %2, %3;" : "=r"(h) : "r"(he), "r"(gapoe), "r"(0)); \
asm("vmax4.s32.s32.s32 %0, %1, %2, %3;" : "=r"(oe) : "r"(oe), "r"(h), "r"(0)); \
asm("vadd4.s32.s32.s32.sat %0, %1, %2, %3;" : "=r"(h) : "r"(hd), "r"(sub), "r"(0)); \
asm("vmax4.s32.s32.s32 %0, %1, %2, %3;" : "=r"(h) : "r"(h), "r"(f), "r"(0)); \
asm("vmax4.s32.s32.s32 %0, %1, %2, %3;" : "=r"(h) : "r"(h), "r"(oe), "r"(0)); \
asm("vmax4.s32.s32.s32 %0, %1, %2, %3;" : "=r"(h) : "r"(h), "r"(0), "r"(0)); \
asm("vmax4.s32.s32.s32 %0, %1, %2, %3;" : "=r"(maxHH) : "r"(maxHH), "r"(h), "r"(0)); \
asm("mov.s32 %0, %1;" : "=r"(hd) : "r"(he));
```

---

<sup>1</sup>PTX defines a virtual machine and virtual ISA for general purpose parallel thread execution. PTX programs are translated at install time to the target hardware instruction set. The PTX to GPU translator and driver enables NVIDIA GPUs to be used as programmable parallel computers [82].

# APPENDIX C

## PERFORMANCE DETAILS OF GPU-ACCELERATED ALIGNMENT TOOLS

**Table C.1:** Performance details of GPU-accelerated alignment software tools determining overlaps between reads in SYN\_1000.fna dataset

Software	Elapsed real time <sup>a</sup>	User time <sup>b</sup>	System time <sup>c</sup>	GCUPS <sup>d</sup>
CUDASW++ 2.0	0 m 16 s	0 m 11 s	0 m 4 s	2.44
G-PAS	0 m 44 s	0 m 33 s	0 m 9 s	0.89
SW-CUDA	5 m 12 s	3 m 34 s	1 m 25 s	0.13
water	344 m 22 s	150 m 37 s	64 m 49 s	< 0.01

<sup>a</sup>The time or difference between the beginning time and an ending time of the program.

<sup>b</sup>Total amount of CPU-time that the process spent in user mode.

<sup>c</sup>Total amount of CPU-time that the process spent in kernel mode.

<sup>d</sup>Giga Cell Updates per Second.

**Table C.2:** Performance details of GPU-accelerated alignment software tools determining overlaps between reads in SYN\_10K.fna dataset

Software	Elapsed real time <sup>a</sup>	User time <sup>b</sup>	System time <sup>c</sup>	GCUPS <sup>d</sup>
CUDASW++ 2.0	23 m 19 s	17 m 16 s	5 m 58 s	2.79
G-PAS	53 m 31 s	38 m 29 s	13 m 14 s	1.21
SW-CUDA	434 m 32 s	308 m 30 s	122 m 19 s	0.15
water	33,024 m 12 s <sup>e</sup>	N/A	N/A	< 0.01

<sup>a</sup>The time or difference between the beginning time and the ending time of the program.

<sup>b</sup>Total amount of CPU-time that the process spent in user mode.

<sup>c</sup>Total amount of CPU-time that the process spent in kernel mode.

<sup>d</sup>Giga Cell Updates per Second.

<sup>e</sup>This value is extrapolated.

**Table C.3:** Performance details of GPU-accelerated alignment software tools determining overlaps between reads in SYN\_100K.fna dataset

Software	Elapsed real time <sup>a</sup>	User time <sup>b</sup>	System time <sup>c</sup>	GCUPS <sup>d</sup>
CUDASW++ 2.0	2235 m 56 s	1714 m 0 s	515 m 309 s	2.89
G-PAS	N/A	N/A	N/A	N/A
SW-CUDA	N/A	N/A	N/A	N/A
water	3,170,304 m 46 s <sup>e</sup>	N/A	N/A	< 0.01

<sup>a</sup>The time or difference between the beginning time and the ending time of the program.

<sup>b</sup>Total amount of CPU-time that the process spent in user mode.

<sup>c</sup>Total amount of CPU-time that the process spent in kernel mode.

<sup>d</sup>Giga Cell Updates per Second.

<sup>e</sup>This value is extrapolated.

**Table C.4:** Performance details of GPU-accelerated alignment software tools determining overlaps between reads in ENV\_1000.fna dataset

Software	Elapsed real time <sup>a</sup>	User time <sup>b</sup>	System time <sup>c</sup>	GCUPS <sup>d</sup>
CUDASW++ 2.0	0 m 13 s	0 m 9 s	0 m 3 s	2.54
G-PAS	0 m 40 s	0 m 27 s	0 m 9 s	0.82
SW-CUDA	5 m 12 s	3 m 23 s	1 m 16 s	0.13
water	341 m 15 s	143 m 54 s	62 m 28 s	< 0.01

<sup>a</sup>The time or difference between the beginning time and the ending time of the program.

<sup>b</sup>Total amount of CPU-time that the process spent in user mode.

<sup>c</sup>Total amount of CPU-time that the process spent in kernel mode.

<sup>d</sup>Giga Cell Updates per Second.

**Table C.5:** Performance details of GPU-accelerated alignment software tools determining overlaps between reads in ENV\_10K.fna dataset

Software	Elapsed real time <sup>a</sup>	User time <sup>b</sup>	System time <sup>c</sup>	GCUPS <sup>d</sup>
CUDASW++ 2.0	21 m 22 s	16 m 12 s	5 m 5 s	2.80
G-PAS	55 m 55 s	0 m 9 s	0 m 3 s	1.07
SW-CUDA	435 m 54 s	310 m 32 s	115 m 43 s	0.14
water	32,736 m 11 s <sup>e</sup>	N/A	N/A	< 0.01

<sup>a</sup>The time or difference between the beginning time and the ending time of the program.

<sup>b</sup>Total amount of CPU-time that the process spent in user mode.

<sup>c</sup>Total amount of CPU-time that the process spent in kernel mode.

<sup>d</sup>Giga Cell Updates per Second.

<sup>e</sup>This value is extrapolated.

**Table C.6:** Performance details of GPU-accelerated alignment software tools determining overlaps between reads in ENV\_100K.fna dataset

Software	Elapsed real time <sup>a</sup>	User time <sup>b</sup>	System time <sup>c</sup>	GCUPS <sup>d</sup>
CUDASW++ 2.0	2165 m 7 s	1646 m 42 s	512 m 15 s	2.89
G-PAS	N/A	N/A	N/A	N/A
SW-CUDA	N/A	N/A	N/A	N/A
water	3,142,656 m 31 s <sup>e</sup>	N/A	N/A	< 0.01

<sup>a</sup>The time or difference between the beginning time and the ending time of the program.

<sup>b</sup>Total amount of CPU-time that the process spent in user mode.

<sup>c</sup>Total amount of CPU-time that the process spent in kernel mode.

<sup>d</sup>Giga Cell Updates per Second.

<sup>e</sup>This value is extrapolated.



**Table C.7:** Performance details of CUDASW++ 3.0, MR-CUDASW ,MIRA, and water. Datasets of sizes of simulated datasets were used for this experiment.

Software	Elapsed real time <sup>a</sup>	User time <sup>b</sup>	System time <sup>c</sup>	GCUPS <sup>d</sup>
SYN_1000.fna				
CUDASW++ 3.0	0 m 7 s	0 m 9 s	0 m 1 s	5.95
MR-CUDASW	0 m 4 s	0 m 11 s	0 m 1 s	9.79
MIRA 4.0	0 m 26 s	0 m 3 s	0 m 4 s	1.50
water	848 m 31 s	311 m 1 s	134 m 13 s	< 0.01
SYN_10K.fna				
CUDASW++ 3.0	1 m 53 s	6 m 20 s	0 m 11 s	34.64
MR-CUDASW	1 m 16 s	1 m 59 s	0 m 7 s	51.51
MIRA 4.0	0 m 58 s	0 m 12 s	0 m 11 s	67.49
water	81,408 m 36 <sup>e</sup>	N/A	N/A	< 0.01
SYN_100K.fna				
CUDASW++ 3.0	102 m 32 s	408 m 18 s	2 m 16 s	63.15
MR-CUDASW	79 m 25 s	145 m 25 s	1 m 37 s	81.53
MIRA 4.0	37 m 6 s	7 m 11 s	2 m 33 s	174.53
water	7,815,168 m 12 s <sup>e</sup>	N/A	N/A	< 0.01

<sup>a</sup>The time or difference between the beginning time and the ending time of the program.

<sup>b</sup>Total amount of CPU-time that the process spent in user mode.

<sup>c</sup>Total amount of CPU-time that the process spent in kernel mode.

<sup>d</sup>Giga Cell Updates per Second.

<sup>e</sup>This value is extrapolated.

**Table C.8:** Performance details of CUDASW++ 3.0, MR-CUDASW ,MIRA, and water. Datasets of various sizes of real WGS dataset were used for this experiment.

Software	Elapsed real time <sup>a</sup>	User time <sup>b</sup>	System time <sup>c</sup>	GCUPS <sup>d</sup>
ENV_1000.fna				
CUDASW++ 3.0	0 m 6 s	0 m 8 s	0 m 1 s	5.51
MR-CUDASW	0 m 4 s	0 m 5 s	0 m 0 s	8.27
MIRA 4.0	0 m 12 s	0 m 0 s	0 m 1 s	2.75
water	835 m 12 s	302 m 8 s	117 m 35 s	< 0.01
ENV_10K.fna				
CUDASW++ 3.0	1 m 33 s	5 m 17 s	0 m 10 s	38.72
MR-CUDASW	1 m 7 s	3 m 51 s	0 m 8 s	53.75
MIRA 4.0	0 m 19 s	0 m 3 s	0 m 4 s	189.55
water	80,160 m 25 s <sup>e</sup>	N/A	N/A	< 0.01
ENV_100K.fna				
CUDASW++ 3.0	104 m 49 s	405 m 54 s	2 m 8 s	59.77
MR-CUDASW	84 m 21 s	326 m 8 s	1 m 50 s	71.72
MIRA 4.0	2 m 49 s	0 m 36 s	0 m 35 s	2224.43
water	7,695,360 m 7 s <sup>e</sup>	N/A	N/A	< 0.01

<sup>a</sup>The time or difference between the beginning time and the ending time of the program.

<sup>b</sup>Total amount of CPU-time that the process spent in user mode.

<sup>c</sup>Total amount of CPU-time that the process spent in kernel mode.

<sup>d</sup>Giga Cell Updates per Second.

<sup>e</sup>This value is extrapolated